UNIVERSITY OF CALIFORNIA

Santa Barbara

MLTP: a Two-Level Thread Library for SMP Linux Machines

A Thesis submitted in partial satisfaction

of the requirements for the degree of

Master of Science

in

Computer Science

by

Michael Dipperstein

September 2000

The Thesis of Michael Dipperstein is approved by

Committee Chairperson

December 2000

September 1, 2000

Acknowledgements

I wish to thank Marco Dominguez-Lerma, Pat Anderson, Patti Gilbert, Anne Lanthorpe, and the rest of the staff at UCSB's Ventura Center for making my remote learning experience as pleasant as possible. If it had not been for their efforts and the Off Campus studies program, I would not have been able to pursue a Master of Computer Science degree.

I would also like to thank Marco Dominguez-Lerma and Mary Jane Archenbronn for their assistance with all the on campus leg work and paper pushing. Their combined efforts have saved me countless hours and allowed me to avoid at least one extra oil change on my car.

I would like to thank my thesis committee of Tao Yang, Omer Egecioglu, and Kevin Almeroth for their editorial input on this thesis and their extraordinary ability to work around their own complex schedule needs as well as mine in order submit this thesis.

I would like to thank those who have helped me, though it is not required by even the loosest interpretation of their job description: Hong Tang and Kai Shen for their technical direction and providing me with the initial idea that became this project. Jeff Koftinoff for developing the jkthreads package and allowing me to modify and redistribute a significant portion of it outside of the initial licensing agreement. David Keppel for allowing free modification and redistribution of his QuickThreads package.

Finally, I would like to thank my wife Terri for encouraging me to go back to school, assisting me with the reproduction of this paper, and tolerating the extra time demands of both my full-time time career and part-time graduate school.

Abstract

MLTP: a Two-Level Thread Library for SMP Linux Machines

by

Michael Dipperstein

This thesis is focused on the design and implementation of an open-source two-level thread package called MLTP for the Linux operating system running on Intel PC SMPs. Kernel threads directly scheduled by multiprocessor OS normally have high context switch cost compared to user-level threads. User-level threads scheduled within a single kernel process are not capable of utilizing multiple processors. Many parallel applications running on SMPs require support for flexible control of kernel and user-level threads and such a package is not available on the Linux operating system. In the thesis, I will discuss an M-to-N architecture for MLTP and a design for efficient synchronization and switch among threads. Such a design allows a multi-thread application to achieve scalability and efficiency in multi-processor environments at a low cost. I will also present performance of MLTP for several micro-benchmarks and applications on Intel Xeon dual and quad-processor SMPs.

Contents

List of Tables

List of Figures

Chapter 1 Introduction

This thesis is focused on the design and implementation of an open-source two-level thread package called MLTP for the Linux operating system running on Intel PC SMPs. Many thread packages recently developed for multi-processor shared memory systems have successfully combined user- and kernel-level threads, gaining the advantages of both thread models while avoiding most of their disadvantages. MLTP brings an efficient and extensible two-level thread package to SMP Linux for use in threaded applications and continued development.

1.1 Background and Motivation

Due to recent developments in the Linux kernel and SMP architecture for Intel x86 processors, Linux Intel PC SMPs have become a viable inexpensive platform for parallel applications. Many parallel applications achieve parallelization using the thread paradigm. While, multi-level threads have demonstrated better performance than either user- or kernel-level threads alone on several non-Linux multiprocessor platforms, Linux thread packages are exclusively user- or kernel-level. [16]

1.1.1 User-Level Threads

User-level threads, often called many-to-one threads, execute in the context of a single process. Traditionally, user-level threads each have their own stack space and possibly some thread specific data space. User-level thread packages are easier to implement then kernel-level threads and generally require less time to switch context between threads. Since user-level threads execute in the context of a process, the kernel is not aware of the existence of user-level threads. When a user-level thread blocks, the whole process, and all threads it runs are also blocked. User-level threads are also limited because they run in the context of a single process, and processes cannot run on more than one processor at a time. Consequently, user-level threads may not take advantage of any other available processors on a multiprocessor machine.

1.1.2 Kernel-Level Threads

Kernel-level threads, often called one-to-one threads, execute each thread in the context of its own process and are visible to the kernel. Since kernel-level threads are processes, they do not suffer the same restrictions that user-level threads suffer. If one kernel-level thread is blocked, it is not necessarily the case that all other kernel-level threads are blocked. Since each kernel-level thread is a process, one kernel-level thread may run on one processor, while another kernel-level thread runs on another processor. All these benefits are not free of cost, kernel-level threads are generally more difficult to implement than user-level threads and require the resources of a process. The time to switch context between kernel-level threads is also generally greater than the time to switch context between user-level threads, since kernel-level thread context switching requires costly entries and exits from the Linux kernel.

Due to the implementation of Linux, there are additional restrictions on the number of process, and therefore kernel-level threads that may run on a machine. The standard Linux distribution only allows for 512 processes. The kernel can be recompiled to allow up to 4090 processes. The 4090 process limit is due to Intel x86 architectural constraints. [10] Though 4090 threads will suffice for most applications, the 4090 thread limit has been problematic to some threaded application developers.

1.2 Contribution

The multi-level thread package (MLTP) contributes an extensible, open source, two-level thread package for SMP Linux on Intel x86 processors. By layering user-level threads on top of kernel-level threads, MLTP is able to provide most of the advantages of both thread user- and kernel-level architectures while limiting the disadvantages. MLTP provides:

- Low per-thread overhead. Each user-level thread requires a small amount of space to store context registers and its own private stack. No kernel resources are utilized by MLTP user-level threads.

- Quick thread context switching. User-level thread context switching under MLTP does not require any kernel intervention, there are no kernel traps or process context switches and rescheduling. User-level context switching does not require computations to be made in order to determine which thread should be executed next, therefore the time required for a context switch does not increase as the number of threads increases.
- Multi-processor utilization. By running user-level threads on top of kernel-level threads, the user-level threads may run on multiple processors and still take advantage of a shared thread space.

MLTP has been created as an open source, extensible, and well commented thread package so that it may serve as a platform for further research in multi-level thread systems in SMP Linux.[1] In an attempt to maintain clarity and portability, MLTP does not utilize any kernel patches to provide a multi-layer threading. It is not the intention of this thesis to be the final authority on the subject of multi-level threads for SMP Linux.

MLTP has been benchmarked using both synthetic and application benchmarks, demonstrating a flexible set of primitives and performance improvements which may exceed ten percent when compared to kernel-level threads alone.

1.3 Organization

The rest of this thesis is organized as follows. The work related to MLTP is discussed in Chapter 2. Chapter 3 discusses the implementation of MLTP; it's architecture, the scheduling used, and the user-level primitives implemented. The performance of MLTP is discussed in Chapter 4. Included in this chapter is a discussion on the purpose of each benchmark and the results obtained. Some directions for future work are provided in Chapter 5, and a conclusion is provided in Chapter 6.

[1] The source code for MLTP is available for download from http://www.cs.ucsb.edu/~mdipper/mltp/mltp-1_00.tar.gz

Chapter 2 Related Work

With the prevalence of the programs that exploit the multi-threaded programming model, a fair amount of effort has been put in to the development of more efficient thread platforms. The effort towards improving the efficiency of thread platforms has been divided between optimizing thread primitives and combining the efficiencies of both user- and kernel-level threads. The Multi-Level Thread Package is more closely related to efforts which attempt to combine the benefits of both user- and kernel-level thread architectures. The subsections which follow discuss some of the different approaches which have integrated user- and kernel-level functionality into a single thread package.

2.1 UW Scheduler Activations

Like MLTP, scheduler activations allow user-level influence over kernel scheduling decisions and allow user-level threads in the same thread space to be executed on multiple processors. [1] To achieve its goals scheduler activation entities are created. The scheduler activations are known by the kernel scheduler, and have the ability to run user-level threads on any available processor. In this respect scheduler activations are similar to MLTP kernel-level threads. When the a scheduling change occurs, an up call is made to a user-level thread scheduler. The thread scheduler then makes the determination of which, if any user-level thread may be executed. When a user-level thread is to be preempted, the user-level thread scheduler may make another thread available for execution. This technique allows for progress of user-level threads while a thread in the same thread space is blocked. While promising results have been achieved using scheduler activations, kernel modifications are required to utilize its architecture.

2.2 Sun Threads

SunOS 5 introduced one of the earliest commercial two-level thread architectures. [13] With the exception of hardware platforms and operating systems the basic architecture of Sun Threads are very similar to MLTP threads. By using a two-layer approach, threaded applications are less effected by scheduler decisions and may operate on multiple processors. The lower layer of Sun Threads are

Light Weight Processes (LWPs), individual processes which share memory with each other. The LWPs serve as "virtual CPUs" on which user-level threads are executed. Under Sun Threads, the kernel's scheduler is responsible for the scheduling of the LWPs, it determines which processor a LWP may run on and when it may run. The LWPs are the responsible for determining which of the which of the user-level threads they will run. Additional support is added which binds a single thread to an LWP. Bound threads may only be executed by a single LWP. If a thread is bound to an LWP, the bound LWP may not execute other threads.

A 1996 study conducted at Brown University concluded that there Sun Threads are generally well behaved with a serious exception. [5] Sun Threads showed poor performance when used with fine grain barrier operations. This poor performance was blamed on the "parking" and "unparking" of LWPs during barrier synchronizations. Section 3.3.4 discusses the implementation of MLTP barriers and why MLTP is not subject to the same problems.

2.3 UIUC Nanothreads

The Nanothreads architecture, is another thread architecture which gains much of its improvements through tighter kernel integration. Like scheduler activations, Nanothreads have achieved much of their performance improvements through exporting scheduling decisions normally made in the kernel to a user-level scheduler. [6] The Nanothreads architecture builds upon scheduler activations by exporting scheduler resource allocation to the user-level as well. Designed for multiprogramming environments, Nanothreads uses a space sharing schedule for Nanothreaded applications. As with scheduler activations, Nanothreads requires kernel modifications to share scheduling decisions with a user-level scheduler.

2.4 UCSB TMPI Threads

The SGI IRIX version of TMPI uses a two-level thread package which provided a basis for much of MLTP. [15] Like MLTP, TMPI Threads layer user-level threads on top of kernel-level threads, or "virtual processors", to allow the kernel's scheduler to distribute threads among processors while maintaining an inexpensive context switching. Having been designed for a multiprogramming environment, TMPI Threads provide an additional element of coordination which restricts the number of virtual processors to be less than or equal to the number of actual processors. The pool of available virtual processors is shared among all TMPI Threaded applications on the system. The addition of virtual processor coordination is discussed in section 5.3.

Chapter 3 Multi-Level Thread Package (MLTP)

This section discusses the design of the Multi-Level Thread Package for SMP Linux. In section 1.2, claims are made indicating that MLTP provides low per-thread overhead, quick thread context switching, and multi-processor utilization. In this section the mechanisms that allow for each of these features will be explained in detail.

3.1 MLTP Architecture

The two levels of the MLTP architecture are the process layer (section 3.1.1) and the user layer (section 3.1.2). The process layer provides a collection of virtual processors (VPs), which are visible to the Linux kernel's scheduler. The user layer provides a collection of user-level threads which are not visible to Linux kernel’s scheduler, but are visible to the VPs.

The Linux kernel is responsible for scheduling VPs and allowing them to run on available CPUs. The Linux kernel is not aware of user-level threads, therefore the virtual processors are responsible for scheduling and running the user-level threads. Figure 1 illustrates the relationship between CPUs, the Linux kernel’s scheduler, VPs, and user-level threads in a single MLTP application. The subsections which follow serve to clarify this relationship.

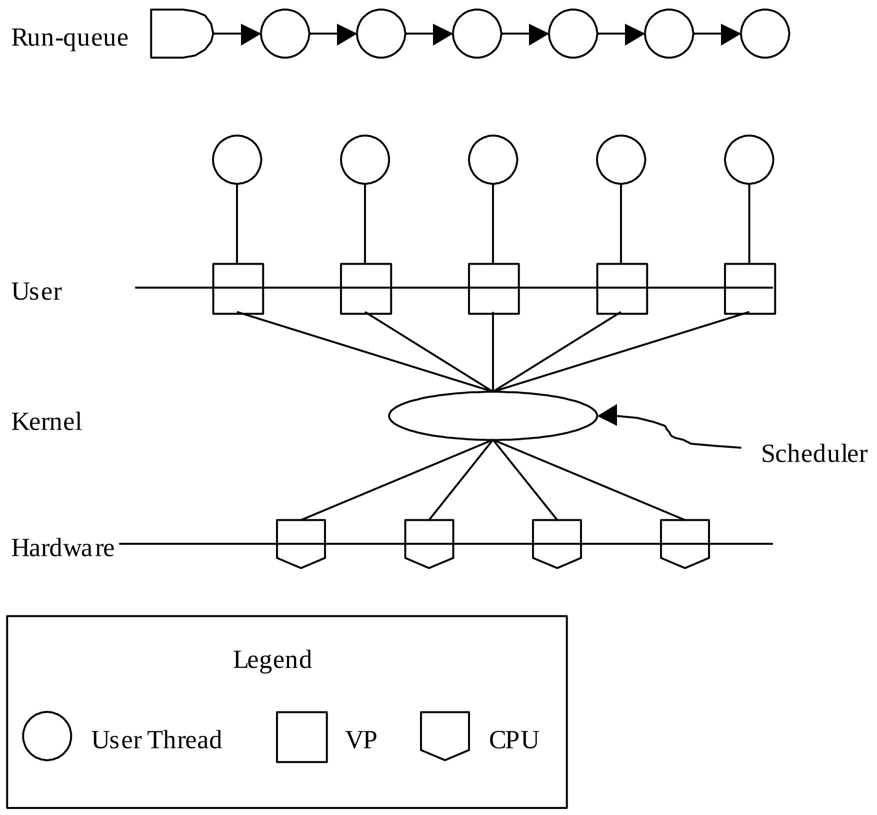


Figure 1 MLTP Application Architecture

3.1.1 Process Layer

The MLTP processes layer consists of a collection of virtual processors (VPs). Like the VPs of TMPI Threads and the virtual CPUs of Sun Threads, MLTP VPs provide a context for running user-level threads. MLTP VPs are Linux kernel-level threads created using a modified version of Jeff Koftinoff's jkthreads package. [9] Earlier versions of MLTP attempted to use the more common LinuxThreads package, which has been closely integrated with the latest Linux C library (glibc), to provide suitable virtual processors. [10] These attempts were abandoned because LinuxThreads are required to maintain the same stack space from creation to termination. When a VP provides a context for running user-level threads, the VP's active stack becomes the stack of the user-level thread.

Each VP is a process cloned from a threaded application's base process, using the Linux **clone()** system call. The **clone()** system call, along with some additional stack manipulation, creates VPs which act much like classical kernel-level threads. MLTP VPs have the following properties:

- Each VP is a Linux process. As with all other Linux processes, MLTP VPs are scheduled by the Linux kernel's scheduler. As processors become available, the Linux kernel's scheduler will use its scheduling algorithm to execute the MLTP VPs. Context switching between VPs and other Linux processes is preemptive.
- Each VP shares resources system resources. The **clone()** system call that is used to create the VPs, creates them with a common memory space and file system. The signal handlers that are in place before the **clone()** call is made are also copied to all VPs.[2]
- Each VP has a unique stack space. As with most thread implementations, MLTP virtual processors have their own stack space allowing independent function call sequences.

Synchronization between VPs is achieved using System V semaphores. Under the current MLTP implementation, a VP will continue to run until, there are more VPs than user-level threads. The VPs which become idle due to an insufficient number of user-level threads will terminate themselves. Though simple, this approach requires that all user-level threads be created before the number of user-level threads drops below the number of VPs. This restriction is not a problem for most applications, however avoided if the VP termination algorithm is modified so that VPs are "parked" until either a user-level thread becomes available to execute, or there are no more user-level threads (section 5.5).

[2] The Linux **clone()** command allows cloned processes to share the same signal handler, but sharing the signal handler will prevent bound threads (section 3.1.2.2) from using customized signal handlers.

One consequence of using cloned processes to serve as virtual processors is that the termination of each of the processes is handled in the signal context of it's parent. In this case the parent is the application's original process. Early tests using code that evolved into some of the samples included with the MLTP release, indicated that allowing the parent process to act as a VP occasionally caused a large increase in the overall application completion time. It is likely that these slowdowns are caused by executions where the signal handler for the parent process was executing while its base level code held a spin lock. Diagnostic code placed in the signal handler indicated that such events did in fact occur during all executions which ran slower than average. However all instances of spin locks being held by the main program while it was in its signal handler did not result in slower execution. The slowdowns, however where alleviated by sleeping the main process during parallel execution. A consequence of this is that an application using N VPs will have N + 1 processes associated with it when all of its VPs are ready.

3.1.2 User Layer

The user layer of MLTP is the portion of the thread package which is called directly by application code; it consists primarily of user-level threads and a run-queue which the are not visible to the Linux kernel's scheduler. The user layer also includes threads which are bound to a specific process and visible to the Linux kernel's scheduler (section 3.1.2.2).

3.1.2.1 Unbound Threads

Unbound threads are derived from David Keppel's QuickThreads user-level thread tools. [8] Unbound threads are user-level threads that run in the context of any VP using their own stack and saved set of context registers. As with classical user-level threads, MLTP user-level threads are not visible to the Linux kernel's scheduler. This allows for cooperative context switching among user-level threads, instead of the preemptive context switching forced by the Linux kernel's scheduler. This also means that there are no kernel traps required to context switch between user-level threads.

Unbound threads are light weight, and do not have their own file system, memory map, or signal handlers; these resources belong to the VP the executing unbound thread. It is the intent of MLTP that an unbound thread be able to run in the context of any VP, to ensure this all virtual processors share the same file system and memory map. As stated in section 3.1.1, it is possible to install different signal handlers in VPs, if this is done, care must be taken to ensure that differences in signal handlers does not effect an unbound thread if it should happen to migrate among VPs during its lifetime.

3.1.2.2 Bound Threads

The concept of bound threads has been borrowed from Sun Threads. [13] Under Sun Threads, a bound thread is a user-level thread that is always executed in the context of the same virtual CPU. MLTP bound threads differ slightly. Like virtual processors, MLTP bound threads are kernel-level threads derived from jkthreads. The jkthreads used to create bound threads are wrapped in a thread structure similar to unbound threads allowing them to function like unbound threads in all aspects except thread scheduling. Since bound threads are processes, they are scheduled by the Linux kernel's scheduler, and therefore have preemptive multitasking and the context switching cost of Linux processes.

Because of their disadvantages, it is not recommended that bound threads be used where unbound threads suffice. Bound threads are intended to serve daemon functions in MLTP threaded applications. They may sleep, waiting on specific events, and then utilize the shared memory space of the threaded application to handle the occurrence of those events.

3.2 Thread Scheduling

Once an application's multi-threaded execution begins, the scheduling of an application's threads becomes two-fold. The virtual processors, used for executing user-level threads, are Linux processes. As with all but real-time SMP Linux processes, MLTP virtual processors are preemptively scheduled based on their “goodness” (priority and factored with some processor affinity adjustments). [3] MLTP does not make any changes to the Linux kernel's scheduler.

Bound user layer threads are scheduled by the Linux kernel's scheduler, in the same fashion as virtual processors and other Linux processes. Unbound threads in user layer are not visible to the Linux kernel's scheduler and therefore must be scheduled by an entity that is aware of their presence. The current implementation uses a cooperative round-robin scheduling technique for unbound user-level threads. All ready user-level threads are placed in a global run-queue, when a virtual processor becomes available to execute a thread, the thread at the head of the run-queue is removed from the queue and executed in the context of the VP. That thread will continue to execute in the context of the same virtual processor until it yields or aborts (section 3.3.2). Figure 2 depicts a user-level thread scheduling scenario with a two VPs and four user-level threads.

The scheduling of unbound threads is round-robin, and utilizes a global run-queue, therefore a thread may migrate among all virtual processors during its lifetime. Currently, no effort is made to provide virtual processor affinity. Section 5.2 discusses potential benefits to be gained by virtual processor affinity and methods that may be used to provide for a greater affinity.

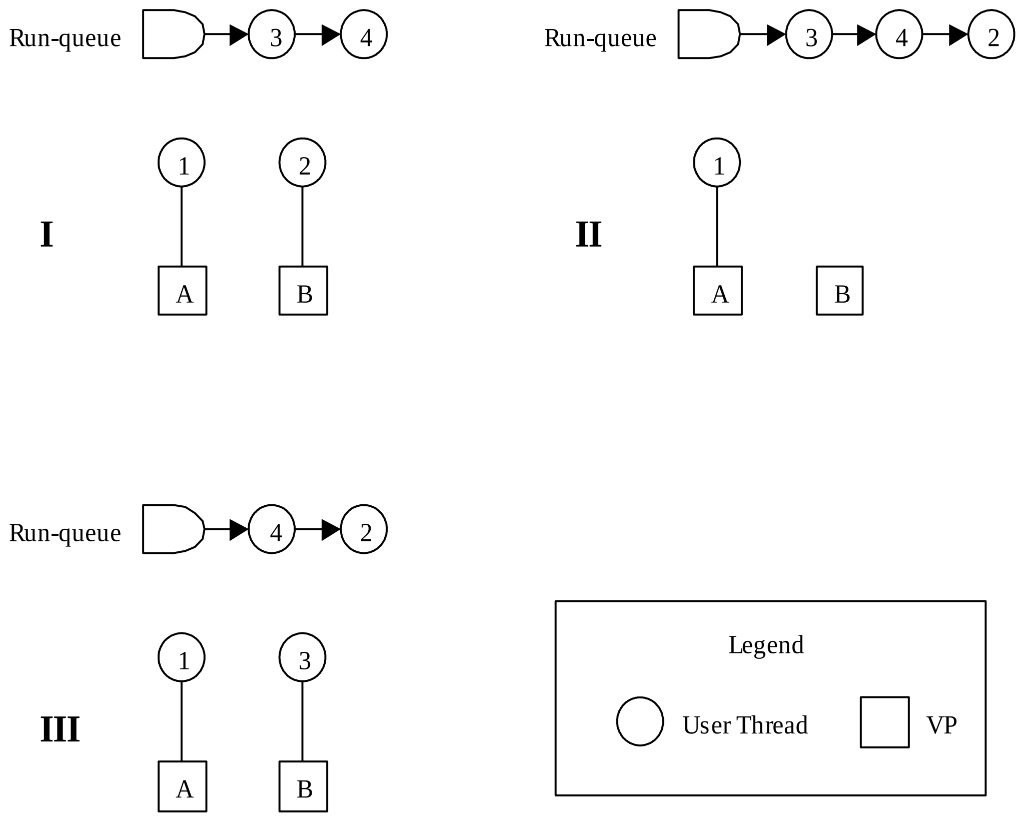


Figure 2 User-Level Thread Scheduling

I Two virtual processes (A and B) are used to run 4 user-level threads (1, 2, 3, and 4). Thread 1 is running in the context of VP A, thread 2 is running in the context of VP B. Threads 3 and 4 are ready in the run-queue. **II** Thread 2 yields and is placed at the end of the run-queue. VP B is now available. **III** Thread 3 is removed from the run-queue and executes in the context of VP B.

3.3 User Level Thread Primitives

MLTP has been designed with a flexible set of user-level thread primitives, capable of supporting the need of most threaded applications. Since MLTP is open source, any additional required primitives may be compiled into the MLTP library. MLTP is built with functions supporting:

- Thread Creation
- Thread Yielding
- Critical Section Locking

- Barrier Synchronization
- Conditional Waiting and Signaling

The remainder of this section discusses each of functions and the details of the primitives that implement them. The source code which implements MLTP is available for download from http://www.cs.ucsb.edu/~mdipper/mltp/mltp-1_00.tar.gz

3.3.1 Thread Creation

MLTP provides for two types of threads, unbound threads (section 3.1.2.1) and bound threads (section 3.1.2.2). Though the user functions which create either type of thread are similar, the results achieved are very different. It is recommended that unbound threads be used for standard application threading, while bound threads be used to perform daemon functions.

3.3.1.1 Unbound Threads

MLTP unbound user-level threads are created using routines derived from David Keppel's QuickThreads library. [8] An artifact of the QuickThreads library is the ability to create threads, passing either a single argument or a variable argument list to the main thread process. MLTP continues to support each of these methods.

```
mltp_t *mltp_create(mltp_userf_t *func, void *p0);
mltp_t *mltp_vcreate(mltp_vuserf_t *func, int nbytes, ...);
```

Regardless of the thread creation method used, each function: allocates and aligns a thread's stack space, pushes the content of the thread's context registers and the address of the thread's main function on the stack, and places the thread at the end of the run-queue. Once created, an unbound thread will not begin to execute until one of the VPs executes it. For most applications it is recommended that all unbound threads be created prior to starting multithreaded execution.

Currently the number of unbound threads are limited to 32767, because they are given an ID that is stored in a signed short. It is possible to increase the number of threads by changing the thread ID to an unsigned long. This will move the limiting factor from the number of thread IDs which may be assigned to the amount of virtual memory available for user-level thread stacks. SMP Linux on x86 processors places a 1GB limit on the amount of virtual memory it may allocate.

3.3.1.2 Bound Threads

MLTP bound threads are created using routines derived from Jeff Koftinoff's jkthreads library. For compatibility with unbound threads, the structure returned by the function which creates bound threads is identical to the thread structure returned for unbound threads.

```
mltp_t *mltp_create_bound(mltp_buserf_t *func, void *p0,
                          int stacksz, mltp_buserf_t *term);
```

When a bound thread is created, the parent process is cloned using the Linux **clone()** system call, and a new stack space is created for the cloned process. The number of bound threads which may be created is limited the number of processes supported by the Linux system. Once created, a bound thread will begin to execute the thread's main function. This differs from unbound threads which will not begin to execute until a VP executes it.

3.3.2 Thread Yielding

Since unbound MLTP threads cooperatively multitask, a mechanism is required for a thread to voluntarily release a VP to another ready thread. MLTP provides three ways for a thread to do this. The thread may abort, yield to the end of the run-queue, or yield to the front of the run-queue.

```
void mltp_abort(void);
void mltp_yield(void);
void mltp_yield_to_first(void);
```

When a unbound thread aborts, it stops and all its allocated resources are freed. The VP which was executing the thread is free to execute another thread. Though this is a necessary function, if it were the only means of releasing a VP, MLTP would be of limited use. For this reason functions which allow a thread to yield a VP to another thread have been provided.

When an unbound thread yields, it is placed on the run-queue. Under the current MLTP round-robin scheduling algorithm, the VP which provided the execution context for the yielded thread will begin to execute the thread at the head of the run-queue. The standard **mltp_yield()** function will place the yielding thread at the end of the run-queue, requiring that all other queued threads be given the opportunity to run before the yielding thread may run again. For cases where a shorter yield is desired, the **mltp_yield_to_first()** function has been provided. **mltp_yield_to_first()** swaps the currently executing thread with the thread at the head of the run-queue. As a consequence of this, the thread that was at the head of the run-queue will be executed on the current VP, while the thread that was executing on the current VP will be placed at the head of the run-queue. Figure 3 illustrates the effects of each yielding primitive on the run-queue and the VP it is executed from.

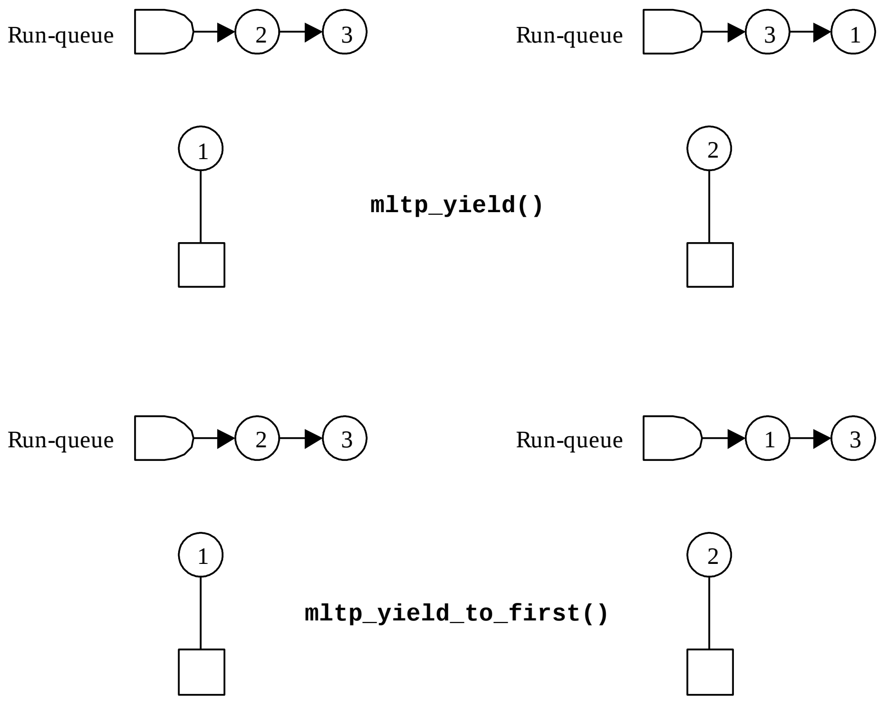


Figure 3 MLTP Unbound Thread Yielding

Bound threads do not run in the context of a virtual processor, the yielding functions provided for unbound threads may not meaningfully support bound threads. For symmetry an **mltp_yield_bound()** macro which yields bound threads has been provided. Instead of yielding a virtual processor to another unbound thread, the yield function for bound threads calls the Linux **sched_yield()** function and yields the CPU to another process.

3.3.3 Critical Section Locking

Critical section locking provides a means of ensuring that only one thread is executing a section sections of code at a time. MLTP provides four types of locks: spin locks, yielding locks (end and front), and semaphore locks. For testing and development, simplicity a fifth lock type, standard lock, is provided. The standard lock type is simply a preprocessor definition. In the current release of MLTP, standard locks are defined to be spin locks. Regardless of the lock type, the same MLTP primitives may be used to acquire and free the locks.

```
void mltp_lock(mltp_lock_t *lock);
void mltp_unlock(mltp_lock_t *lock);
```

3.3.3.1 Spin locks

It is generally the case that in low contention environments, spin locks provide an efficient method of obtaining mutual exclusion. [11] MLTP spin locks are ticket-waiter locks. Each spin lock has a next available ticket, and now serving counter associated with it. Figure 4 lists the code for obtaining a spin lock. The actual code is comprised of at least two phases; lines 4 through 7 attempt to atomically acquire a next available ticket, once the ticket has been acquired line 22 spins until the now serving value matches that of the ticket acquired. Requiring the use of a ticket serves two purposes, the first is that it provides an order in which a lock will be acquired. The first thread to acquire a ticket will be the first to acquire the lock. The second purpose served by the ticket is less obvious. The **mltp_compare_and_swap()** primitive used to obtain the tick is atomic, however it requires multiple bus cycles to complete. To ensure atomicity, the memory bus is locked, preventing other memory writes during a portion of the compare and swap. A simple one phase test and set spin lock requires the same memory bus locking, however the period of bus locking would last for the entire lock acquisition phase, including spin waiting. By using a ticket style of lock acquisition, the memory bus is only locked during the ticket acquisition, not while the rest of the spin waiting occurs.

A conditionally compiled random back-off phase (line 8 through 18) may be included in the ticket acquisition phase, this section is intended to provide some relief to the memory bus when a lock is under heavy contention, however under tests with up to 16 lock waiters, no consistently measurable difference was observed.

```
01: volatile unsigned int ticket;
02:
03: /* get ticket */
04: for (ticket = lock->next_available;
05:     !(mltp_compare_and_swap(ticket, (ticket + 1),
06:         &(lock->next_available)));
07:     ticket = lock->next_available)
08: #ifdef IDLE_SPIN
09: {
10:     int i = 1 + (int)(10.0 * rand() / (RAND_MAX + 1.0));
11:
12:     /* spin up to 10 cycles without locking the memory bus */
13:     while(i)
14:     {
15:         i--;
16:     }
17: }
18: #endif
19:     ;
20:
21: /* spin until ticket is being served */
22: while (ticket != lock->now_serving);
```

Figure 4 Spin Lock Acquisition

When the thread that acquired the lock no longer needs it, the lock is released by incrementing it's now serving counter. Lock releasing need not be atomic, since only one thread may have the lock to release at any given time.

3.3.3.2 Yielding Locks

MLTP provides two types of yielding locks, one which places the yielding thread at the end of the run-queue and another which places the yielding thread at the head of the run-queue. Figure 5 lists the code section used to acquire a yielding lock. The current implementation of either style of yielding lock only makes one attempt to acquire a lock before it yields its VP to another thread, when the thread is restarted it will again make a single attempt to acquire the lock. This scheme does not provide for any ordering of acquisitions with respect to the order they are first attempted. Section 5.1 discusses other options which may be used to vary the number of acquisition attempts prior to yielding.

Lock acquisition is atomic, using the **mltp_compare_and_swap()** primitive (line 2), to ensure that only one thread acquires the lock. The lock release is not required to be atomic since only the thread holding the lock may release it.

```
01: /* block on the run queue if lock is not available */
02: while (!mltp_compare_and_swap(0, 1, &(lock->now_serving)))
03: {
04:     if (lock->lock_class == MLTP_LOCK_BLOCK_FRONT)
05:     {
06:         mltp_yield_to_first();
07:     }
08:     else
09:     {
00:         mltp_yield();
01:     }
02: }
```

Figure 5 Yielding Lock Acquisition

Since yielding locks yield the waiting thread on the run-queue, neither the code which implements the run-queue nor bound threads may utilize yielding locks. Bound threads may not be placed in the run-queue since they are scheduled by the Linux kernel's scheduler. The run-queue can not be implemented using yielding locks, because a thread waiting to acquire the queue lock will have nowhere to yield.

3.3.3.3 Semaphore Locks

Semaphore locks are another type of yielding lock. The yielding lock described in section 3.3.3.2 yields a virtual processor to another thread. Semaphore locks yield a CPU to another process. A semaphore lock utilizes System V semaphores which, under Linux, cause a process to sleep until the conditions of the semaphore are met. Under conditions where the number of process a machine is running does not exceed the number of processors available, semaphore locks provide unnecessary overhead. However, when the number of processes exceeds the number of processors, semaphore locks may provide improved performance. An example of such a case is provide in section 4.4.

3.3.4 Barrier Synchronization

MLTP barriers provide a means ensuring that all threads reach a synchronization point before they are allowed to continue processing. Figure 6 lists the source code for implementing barriers under MLTP. Lines 10 though 18 handle the case where the thread entering the barrier is the last thread required to enter the barrier. Lines 20 through 27 handle the case where there are still other threads required to enter the barrier.

In his study of the performance of M×N Sun Threads, Cantrill found that the model performed poorly in the presence of fine grained barriers. [5] This performance degradation was attributed to the overhead of parking and unparking of virtual CPUs during the barrier event. MLTP barriers avoid this degradation by allowing VPs to continue to execute other user-level threads without having to park. When a user-level thread blocks on a barrier, it is placed on the run-queue. The VP which provided an execution context for the blocked thread may then execute the thread at the head of the run-queue. At no time is a VP parked idly.

```
01: int episode;
02:
03: /* obtain barrier lock */
04: mltp_lock(&(barrier->lock));
05: episode = barrier->episode;
06:
07: /* increment number of threads waiting on this barrier */
08: barrier->waiters++;
09:
10: if (barrier->waiters == count)
11: {
12:     /* this is the last thread required to enter the barrier */
13:     /* start new episode */
14:     barrier->episode++;
15:     barrier->waiters = 0;
16:
17:     mltp_unlock(&(barrier->lock));
18: }
```

```
else
{
    mltp_unlock(&(barrier->lock));

    /* wait until each thread hits the barrier */
    while (episode == barrier->episode)
    {
        mltp_yield();
    }
}
```

Figure 6 MLTP Barriers

It should be noted that since barrier blocking occurs on the run-queue, bound threads may not participate in barriers.

3.3.5 Conditional Waiting and Signaling

Under MLTP, conditional waiting is performed much like it is for many other thread packages. Each condition has its own queue. An unbound thread may enter a conditional wait, where it will be placed on the queue associated with the specified condition. The VP which provided the execution context for the queued thread, is then free to provide the execution context for another unbound thread.

```
void mltp_cond_wait(mltp_cond_t *cond);
```

Threads not in the conditional queue are provided two methods to communicate with the conditional queue, they may either signal or broadcast to it. Signaling a conditional queue places the thread at the head of the conditional queue at the end of the run-queue, while broadcasting to a conditional queue places the entire contents of the queue at the end of the run-queue. Neither signaling or broadcasting have any effects on threads which enter the conditional queue after they occur.

```
void mltp_cond_signal(mltp_cond_t *cond);
void mltp_cond_broadcast(mltp_cond_t *cond);
```

Both bound and unbound MLTP threads may signal or broadcast a condition, however only unbound MLTP threads may wait on a condition. Bound MLTP threads may not wait in a queuing structure, since they are scheduled by the Linux kernel's scheduler.

Chapter 4 MLTP Performance

This section discusses performance of the Multi-Level Thread Package for SMP Linux. In section 1.2, claims are made indicating that MLTP provides low per-thread overhead, quick thread context switching, and multi-processor utilization. In this section both synthetic and application benchmarks are utilized to validate these claims.[3] A synthetic benchmark has been created to measure the cost of context switching (section 4.1), while the applications selected are intended to provide a sample of the tasks threaded applications commonly perform with varying memory and synchronization requirements. A Pi approximation benchmark is presented. It requires little memory and little synchronization (section 4.2). Two matrix to matrix multiplication benchmarks which are capable of using large amounts of memory with little or no synchronization are utilized (section 4.3). And an ocean current which simulation requires large amounts of synchronization using variable amounts of memory is utilized as another benchmark (section 4.4).

The machines used to benchmark MLTP have two or four processors on Intel Xeon motherboards. The four processor machines have four 500MHz Intel Pentium III processors with 512KB of cache and 1GB of local RAM. The two processor machines have two 400MHz Intel Pentium II processors with 512KB of cache and 512MB of local RAM. All of the machines are running Linux kernel version 2.2.15. All measurements were performed on machines with no other active user processes.

For each of the benchmarks below, jkthreads were used as the one-to-one (kernel-level) thread that MLTP performance is compared against. The results provided for each of the benchmarks is an averaging of 10 consecutive executions of the benchmarks on the same machine with the same parameters. The timing data from each of the individual executions is presented in spreadsheet format in Appendix A.

[3] The source code for all of the MLTP benchmarks are available for download at http://www.cs.ucsb.edu/~mdipper/mltp/benchmark.tar.gz

4.1 Context Switching

One of the goals of MLTP is to provide a thread package with an inexpensive thread context switch. In order to determine the cost of a thread context switch and compare it to the cost of a process (or kernel-level thread) context switch, simple programs that do little more than context switch were created. The number of user-level and total system CPU instruction cycles were measured using the Performance Counter Library (PCL). [4] The measurements that PCL provides are not capable of following a process through CPU migration, so care must be taken when running the context switching benchmark. If it is executed in a multiprogramming environment, the CPU cycles per context switch might not represent the actual cost.

As one would expect, MLTP user-level thread context switching incurs most of its cost through user-level code, while Linux process context switching incurs most of its cost in kernel-level code. Figure 7 depicts the number of CPU cycles verses the number of context switches. Fortunately, the total number of instruction cycles per a context switch is less than half that for MLTP threads as it is for Linux processes (378 instruction cycles vs. 830 instruction cycles). With a processor operating at 500MHz, this comes out to a savings of about 2µs per a context switch.

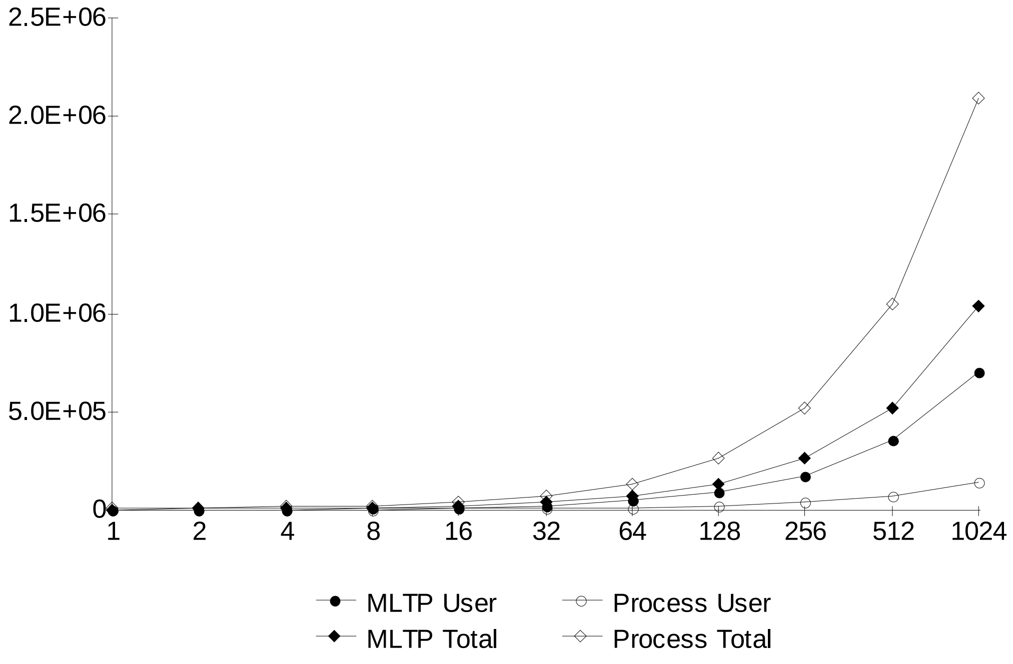


Figure 7 CPU Cycles vs. Context Switches

To fully account for context switch cost, there are two other factors which should be considered: the cost of refreshing the memory caches and page maps, and the cost of calculating which thread is executed after the context switch. Since both user- and kernel-level threads use the same memory space, the cost of updating memory caches and page maps is the same for a threaded application context switching under the same conditions regardless of the type of thread used. The cost of determining the next thread to execute is a little different and not reflected in the benchmark code. MLTP threads use a strict round-robin schedule, the thread switching out is placed at the end of the run-queue, and the thread at the head of the run-queue is removed from the run-queue. At the time of the context switch, both the head and the tail of the run-queue are known and there is no need to perform any calculations. Kernel-level threads are Linux processes, and when a processor becomes available to run a Linux process, the Linux kernel's scheduler calculates the "goodness" of all ready processes. The ready process with the highest "goodness" is the next to execute. [3] Though a simple calculation, increasing the number of ready processes increases the number of CPU cycles required to determine which Linux process is the next to run. Thus increasing the number of kernel-level threads increases the number of CPU cycles required to switch thread context, but increasing the number of user-level threads does not increase the amount of CPU cycles required to switch thread context.

4.2 Pi Approximation

One of the more common examples of threaded code is a program that approximates the value of Pi by using the rectangle rule for integration to compute the area inside the unit circle. This and any other integration by the rectangle rule is trivially parallelizable. Individual rectangles may be distributed among the threads to sum into a local sum, and then by adding each thread's local sum the result of the integration may be obtained. The distribution of rectangular regions by the Pi approximation is fixed, based upon the total number of threads. When a thread is active it sums all of the regions that it is responsible for, then acquires a lock for the global sum, adds it's local sum to the global sum, releases the lock and exits. Besides the acquisition of a lock for the global sum, there are no synchronizations or forced context switches required by the Pi approximation. The lack of context switching and synchronization, make the Pi approximation a poor choice for an application that may benefit from the use of MLTP. This application has been included in the benchmark suite, in order to demonstrate that MLTP performs reasonably well with applications that it was not designed to benefit.

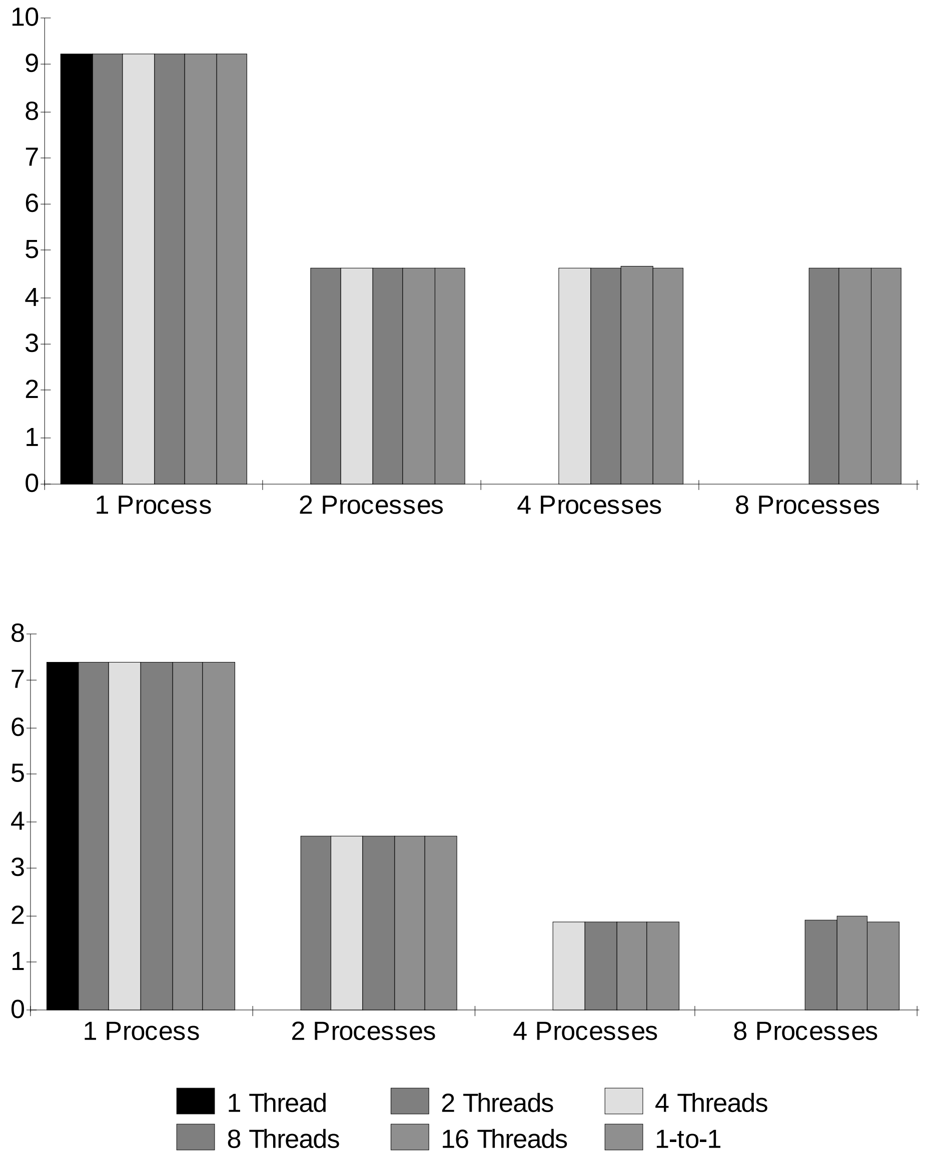


Figure 8 Pi Approximation Computation Time

The above graphs illustrate the number of seconds required to approximate Pi using a 10,000,000 slice approximation. The top graph shows the results for a two processor machine; the bottom graph shows the results for a four processor machine.

Figure 8 depicts the amount of time required to approximate Pi using 10,000,000 rectangular slices.

The only significance of the number 10,000,000 is that it required the Pi approximation to run long enough to obtain measurable differences between execution scenarios. In all cases, the performance of MLTP was close to that of the kernel-level threads utilizing the same number of processors. While the number of user-level threads had little impact on the program's performance, the number of processes certainly did. Using either MLTP or kernel-level threads optimal results were achieved when the number of processes was equal to the number of processors. In many instances MLTP with one thread per a virtual processor performed slightly better than kernel-level threads. The likely explanation for this phenomena is that MLTP uses spin locks, which are capable of providing better performance in low contention environments than the System V semaphores used by the kernel-level threads.

4.3 Matrix to Matrix Multiplication

A technique commonly used to parallelize the multiplication of two N×N matrices is to divide the matrices into submatrices and perform the multiplication of each of the submatrices in parallel. The submatrix multiplications may be performed in any order so long as all the required combinations have been multiplied together. The ability to schedule the multiplications in any order has lead to a number of scheduling approaches. Two of the more common approaches are dynamic scheduling of submatrix multiplication and fixed scheduling of contiguous submatrix multiplications.

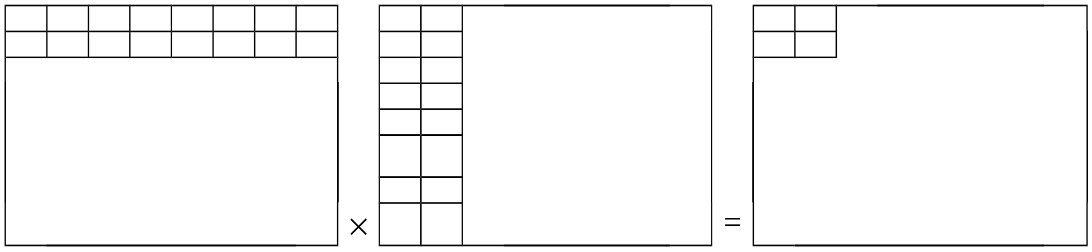


Figure 9 Submatrix Multiplication

This figure illustrates how the upper left 2×2 submatrix of the matrix resulting from the multiplication of two 8×8 matrices may be computed by multiplying an 8×2 submatrix with a 2×8 submatrix.

4.3.1 Matrix to Matrix Multiplication with Dynamic Scheduling

The goal behind dynamic scheduling of submatrix multiplications is to assign a thread a small set of submatrices to multiply when ever a thread is available to perform multiplications. Since threaded processing makes no guarantees to the amount of CPU time a thread will be allowed to have, the dynamic scheduling approach is an attempt to schedule more multiplications with threads that end up with more CPU time. One variation of dynamic scheduling is dynamic self-scheduling, the advantage to self-scheduling is that it does not require an outside scheduling entity. When a thread becomes available to do some processing, it locks the collection of available multiplications, schedules a group of them for itself, and unlocks the collection. A single lock is required for the self-scheduling algorithm, and no locks are required for the actual matrix multiplication.

Though dynamic self-scheduling works well in a preemptive multitasking environment, it does not serve well in a cooperative multitasking environment. There is no reason for an active thread to yield and allow an inactive thread to do some multiplications. For this reason, the self-scheduling matrix to matrix multiplication benchmark only measures the performance of MLTP using one user-level thread per VP.

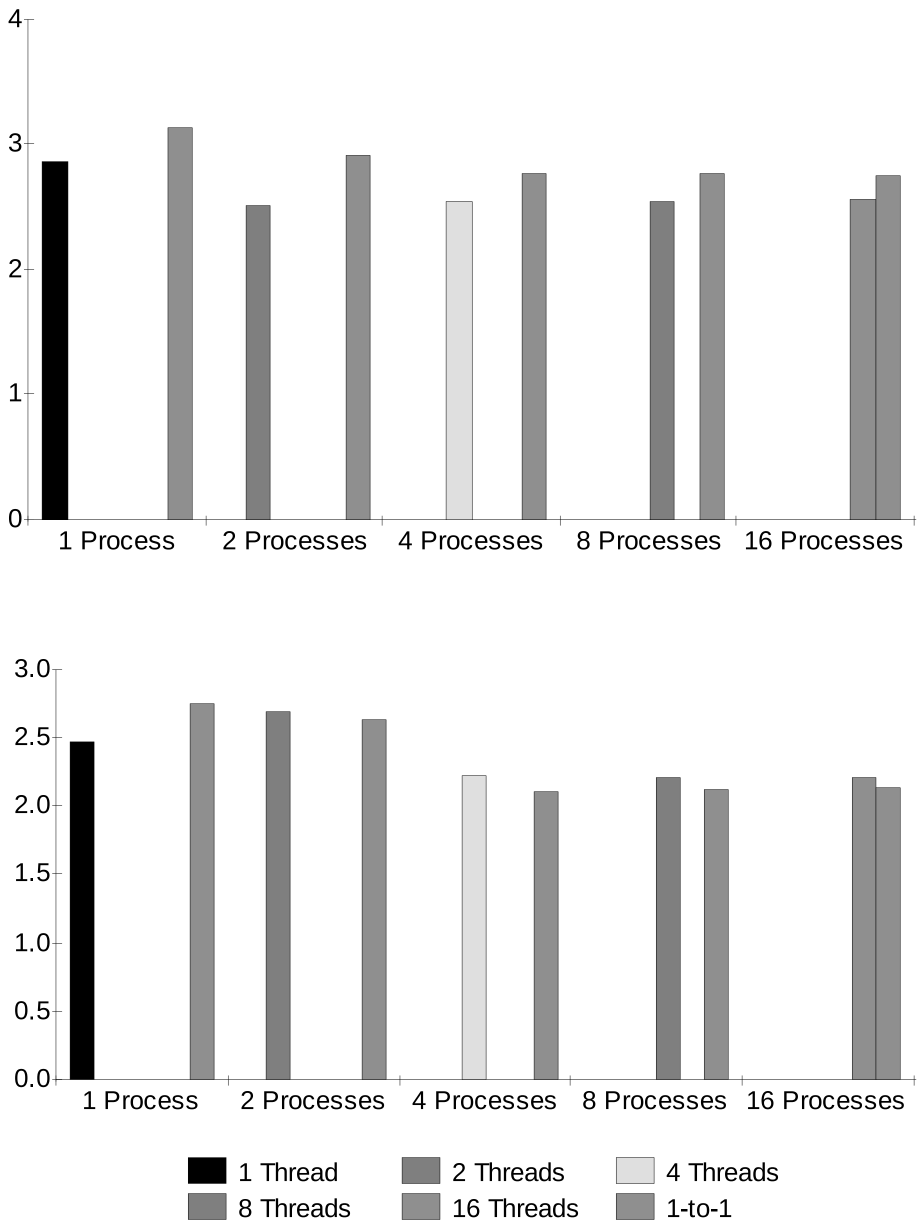


Figure 10 Dynamically Schedule 256x256 Matrix Multiplication

The above graphs illustrate the number of seconds required to multiply two 256 x256 matrices. The top graph shows the results for a two processor machine; the bottom graph shows the results for a four processor machine.

Figure 10 depicts the results of the matrix to matrix multiplication with dynamic scheduling. A

256×256 matrix of double precision values was selected for this test, because the cache of the benchmark machines is large enough to hold the contents used for a single submatrix multiplication, but it is not large enough to hold the contents used to multiply both matrices in their entirety. Based upon empirical results achieved, there is no clear-cut advantage to using either a kernel-level thread package or MLTP. On the two CPU machine, MLTP outperforms kernel-level threads, however more often then not, the opposite is true for the four CPU machine. However intuition would still indicate that with a limited number of synchronizations and no forced context switching, kernel-level threads should be better suited for the task.

4.3.2 Matrix to Matrix Multiplication with Fixed Scheduling

Fixed scheduled matrix to matrix multiplication attempts to distribute the work load evenly among threads. Algorithms with low computation time are used to determine which submatrix multiplications a thread is responsible and no locks, synchronizations, or external schedulers are required. The scheduling algorithm used by this MLTP benchmark divides all the multiplications up into equal size sequential groups by thread ID. A thread then performs all the multiplications assigned to its ID. Once a thread completes all of its assigned multiplications, it terminates. Fixed scheduling prevents a single MLTP user-level thread from performing all the processing for a single VP.

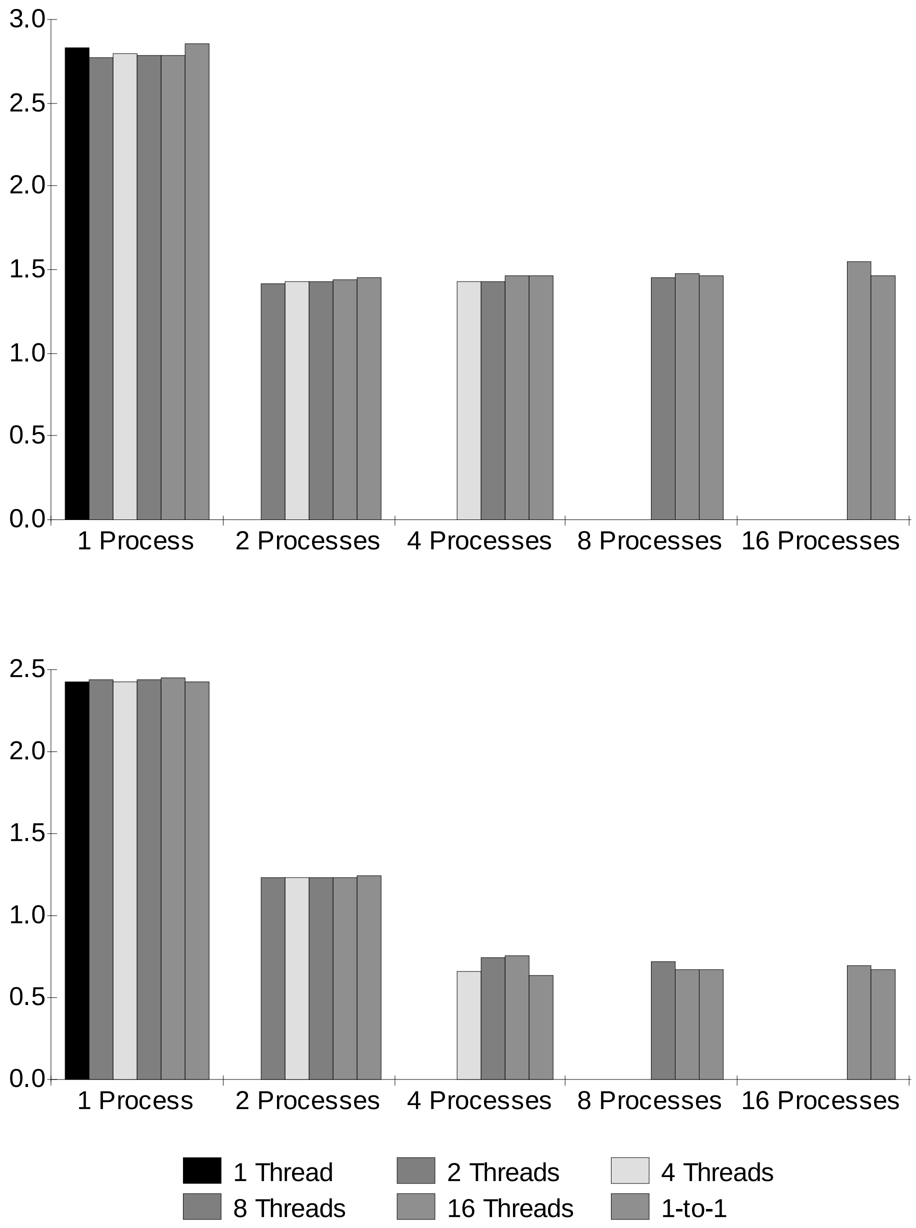


Figure 11 256x256 Matrix Multiplication - Fixed Schedule

The above graphs illustrate the number of seconds required to multiply two 256 x256 matrices. The top graph shows the results for a two processor machine; the bottom graph shows the results for a four processor machine.

Figure 11 depicts the results of the matrix to matrix multiplication with fixed scheduling. A 256×256 matrix of double precision values was selected for this test, because the cache of the benchmark machines is large enough to hold the contents used for a single submatrix multiplication, but it is not large enough to hold the contents used to multiply both matrices in their entirety. On the two CPU machine, MLTP outperforms kernel-level threads, however more often then not, the opposite is true for the four CPU machine. Executions with a large number of threads, on a four processor machine loose efficiency. Its possible that the loss of efficiency may be due to processor migration of the VPs, which could make the RAM cache less effective. Unfortunately processes are required to monitor for CPU migration, and those monitoring processes can themselves increase the likelihood that CPU migration will occur.

Based upon empirical results achieved, there is no clear-cut advantage to using either a kernel-level thread package or MLTP, however intuition would still indicate that with a no synchronizations and no forced context switching, kernel-level threads should be better suited for the task.

4.4 Ocean Simulation

The ocean simulation benchmark has been ported from the Stanford Parallel Applications for Shared Memory (SPLASH-2) ocean current simulation. [18] It computes the current flow for ocean layers given a set of boundaries and initial conditions. An ocean region is divided into a grid of layers and a streamfunction is used to determine how each region effects its neighbor. After a series of iterations, the ocean region will stabilize and an answer will be provided. The ocean simulations is part of a larger class of grid solvers, in which the problem is divided into a grid with initial values and a function that describes how each grid location effects it neighbor. In some cases boundary conditions may apply to the outer grids.

The ocean simulation has 10 synchronization phases in each of its iterations and relies heavily upon barrier synchronizations. Figure 12 depicts the synchronization phases of each iteration and the calculations that lead up to them.

Put Laplacian of Ψ_L in $W1_L$	Put Laplacian of Ψ_3 in $W1_3$	Copy Ψ_L, Ψ_3 to T_L, T_3	Put Ψ_L - Ψ_3 in W2	Put computed Ψ_2 vals in W2	Initialize γ_a and γ_b
Add f values to columns of $W1_L$ and $W1_3$		Copy Ψ_{LM}, Ψ_{3M} into Ψ_L, Ψ_3			Put Lapacian of Ψ_{LM}, Ψ_{3M} in $W7_{L3}$
Put Jacobians of ($W1_L$, $T1_L$), ($W1_3$, $T1_3$) in $W5_l$, $W5_3$		Copy T_L, T_3 into Ψ_{LM}, Ψ_{3M}			Put Lapacian of $W7_{L3}$ in $W4_{L3}$
			Put Jacobian of (W2, W3) in W6		Put Lapacian of $W4_{L3}$ in $W7_{L3}$
Update the γ expressions					
Solve the equations for Ψ_a and put the result in γ_a					
Compute the integral of Ψ_a					
Compute $\Psi = \Psi_a + C(t)\Psi_b$ (Note: Ψ_a and now Ψ are maintained in the γ_a matrix)			Solve the equation for Φ and put result in γ_b		
Use Ψ_a and Φ to update Ψ_L and Ψ_3					
Update streamfunction running sums determine when to stop running program					

Note: Horizontal lines represent synchronization points among all processes, and vertical lines vertical lines spanning phases demarcate threads of dependence.

Figure 12 Ocean Simulation Phases and Barriers

For this benchmark a 258×258 grid ocean, stored as a matrix of double precision values, was utilized. The 258×258 ocean was selected, because a valid solution for the 258×258 ocean is distributed with the SPLASH-2 library. The 258×258 ocean also has the benefit of being too large for all of its calculations to be performed in cached RAM.

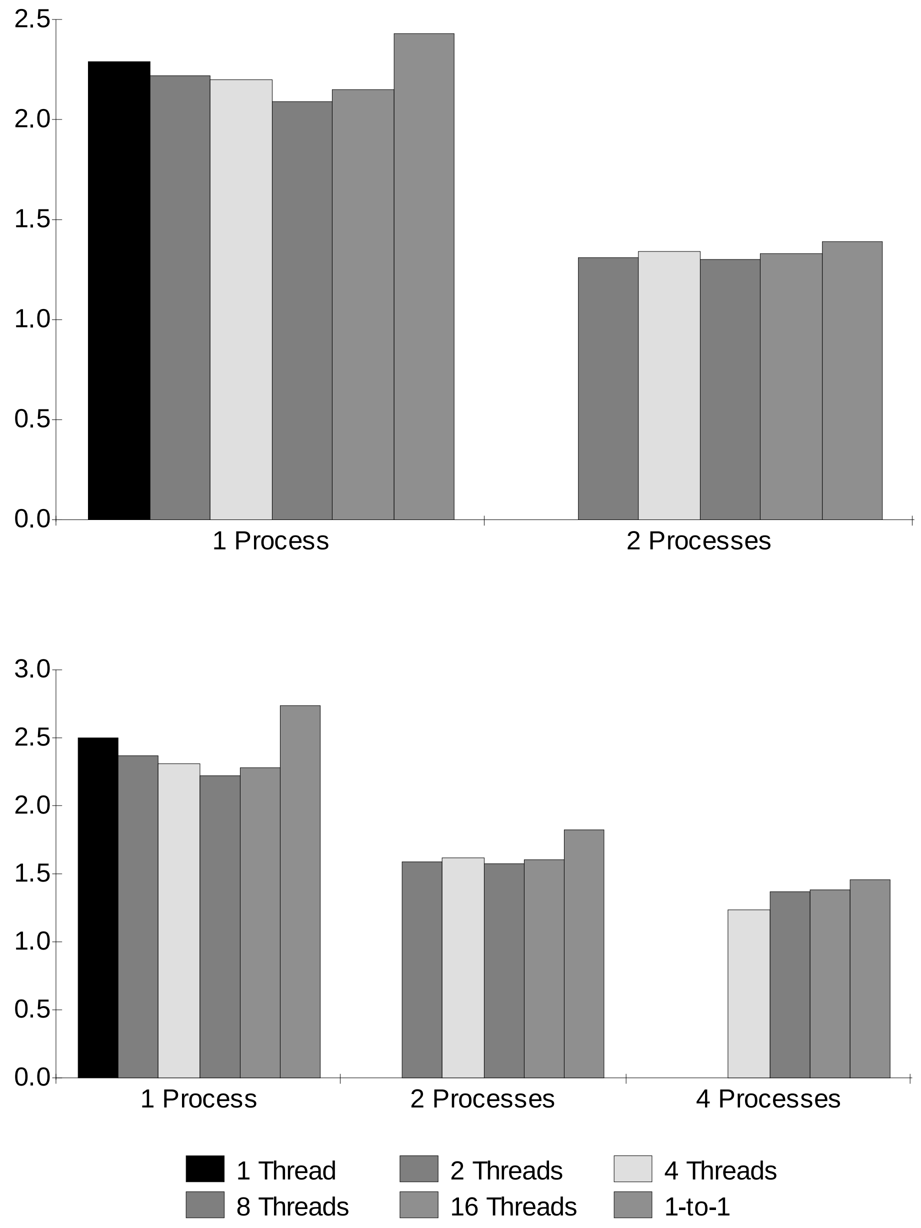
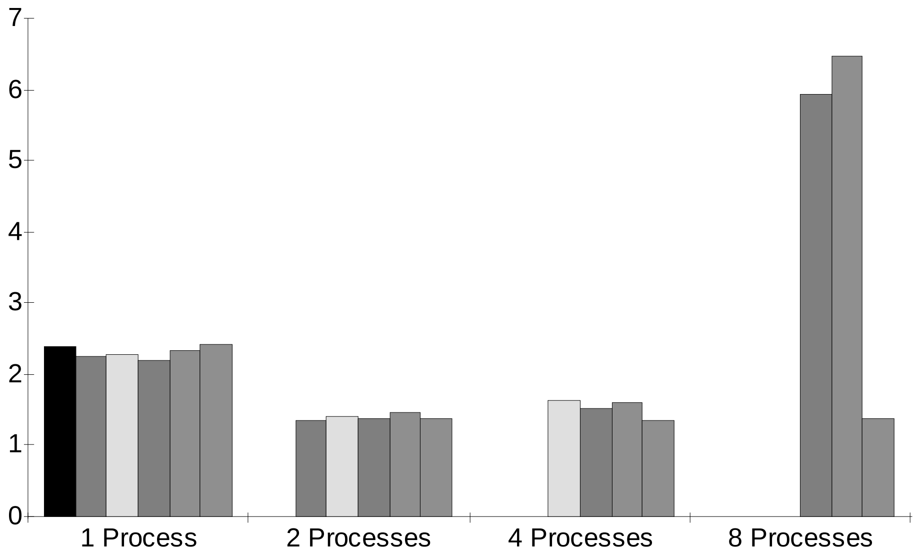


Figure 13 SPLASH-2 Ocean Using Spin Locks

The above graphs illustrate the number of seconds required to compute a solution to a 258 x258 ocean simulation using spin locks. The top graph shows the results for a two processor machine; the bottom graph shows the results for a four processor machine.

The initial MLTP Ocean benchmark used spin locks. The results of the benchmark are depicted in Figure 13. For cases where the number of virtual processors (processes) does not exceed the number of CPUs, MLTP dramatically outperforms kernel-level threads. MLTP typically performs best when there are eight user-level threads. Though, there were no scenarios where all eight user-level threads could run in parallel, a performance gain was still achieved. This is gain occurs because the larger number of user-level threads allows for a more concentrated set of calculations which may take advantage of cached RAM. With eight user-level threads, the synchronization requirements of the extra threads is still less than the performance benefits gained by a larger percent of cache hits. Once 16 threads are added, the synchronization overhead, out weighs the benefits of a large number of cache hits.

Though MLTP performs well when the number of VPs does not exceed the number of processors, it performed poorly when the number of VPs exceeded the number of processors. It takes an average of 448 seconds for MLTP to complete the ocean simulation on a two processor machine with four VPs and four user-level threads. Further study revealed that the slowdown is a result of multiple occurrences where the VP running a lock-holding thread is forced to idle, while the Linux kernel's scheduler gives its CPU to another VP. To combat this, MLTP was recompiled replacing spin locks with semaphore locks. The semaphore locks have higher overhead, but cause the waiting process to sleep, allowing the lock holding processes to run again and complete its task. Figure 14 depicts the results of the ocean simulation using MLTP with semaphore locks. This solution causes MLTP to perform worse than kernel-level threads in most cases. However, it did greatly improve upon the performance of MLTP when the number of VPs exceeds the number of processors. Section 5.1 purposes some alternate implementation which may allow MLTP to perform reasonably without regard to the number of VPs.



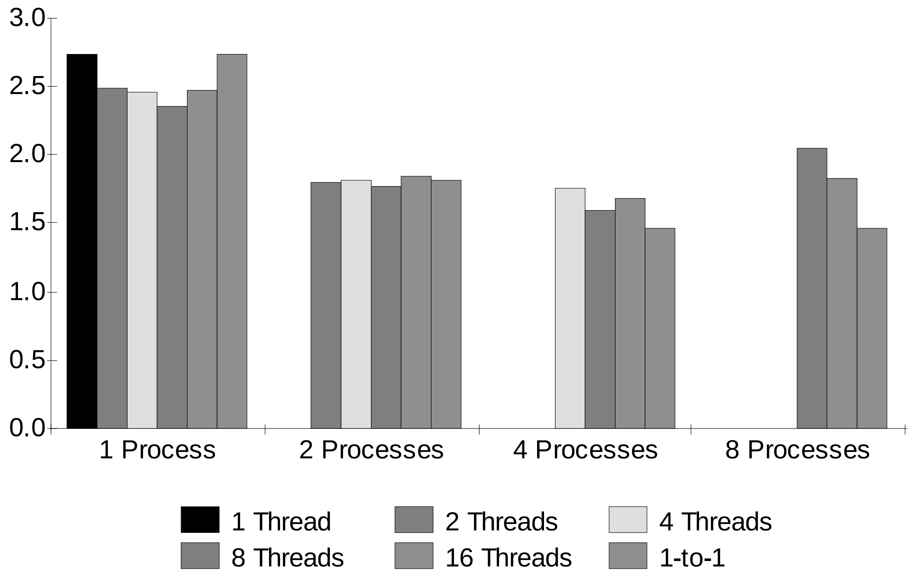


Figure 14 SPLASH-2 Ocean Using Semaphore Locks

The above graphs illustrate the number of seconds required to compute a solution to a 258 x258 ocean simulation using semaphore locks. The top graph shows the results for a two processor machine; the bottom graph shows the results for a four processor machine.

Chapter 5 Future Directions

Throughout this thesis, suggestions were made which indicated that there are still other areas which may be explored and developed. This section discusses the suggestions in greater detail, provides some rational for each suggestion, and includes some guidance which may easy the implementation of each suggestion.

5.1 Critical Section Lock Optimizations

The current implementation of MLTP provides four different approaches to critical section locking. It is the responsibility of the application developer to choose the lock type to be used. As demonstrated by the ocean simulation benchmark (section 4.4), the performance of a lock type may vary substantially with the conditions of lock utilization.

The current spin lock implementation provides for efficient locking under conditions of low contention, but performs poorly when the virtual processor running the lock-holding thread is made idle. Under these conditions, a thread attempting to acquire a lock may spin actively while the thread holding the lock remains idle. Lock performance under this condition would improve greatly if the spinning thread were instead made idle; allowing the lock-holding thread to continue its processing.

Semaphore locks will handle events conditions where the lock-holding thread is idle by sleeping the virtual processor of running the thread attempting to acquire the lock. This makes a CPU available for an idle process, and may reduce the amount of time required before the process with the lock-holding thread runs. A disadvantage to semaphore locks is that they context switch an entire virtual processor, leaving making it unavailable to run other threads. Another disadvantage of semaphore locks is that the context switch will occur anytime a lock is unavailable, regardless of the duration of unavailability.

Yielding locks switch user-level thread context, allowing other user-level threads to progress, while the blocked thread idles on the run-queue. However, yielding locks do not make a processor available for an idle VP running a lock-holding thread. Like semaphore locks, yielding locks also context switch anytime a lock is unavailable, regardless of the duration of unavailability.

Traditionally the problem of lock waiting forcing a context switch at times a short wait may have sufficed, has been addressed using back-off techniques. Back-off techniques provide for a period of spinning, then a period of back-off (yielding or sleeping). These periods repeat until a lock is finally acquired. The problem with using back-off techniques is that optimal or near optimal results may only be obtained if some characteristics of the system are know prior to compilation.

If the characteristics of the application being developed are understood at compile time, applications may be constructed using locks that can provide optimal performance. However, it is typical that runtime environment of an application is either not understood or known to vary. This especially happens in multiprogramming environments. There have been several efforts made to combine spinning and signal waiting, or back-off in a locking technique, however usage of these methods is determined at compile time and differ in degree of effectiveness, depending upon the actual runtime environment. A technique has been purposed to dynamically (reactively) select a lock's synchronization protocol and waiting mechanism at runtime. [11] This technique claims near optimal performance compared to off line selection of the best lock type. The two-layered architecture of MLTP makes the implementation of reactive locks (or any other lock with a waiting mechanism) less than straight forward. Under MLTP, waiting in the form of spinning or signaling may either occur at the user-level, or the kernel-level. Spinning at either level ties up both the user-level thread and the virtual processor. Waiting on a signal at user-level allows a virtual processor to run another user-level thread, while waiting on a signal at the kernel-level, frees up a CPU to run another virtual processor. Prior to implementing reactive locks, these issues should be researched.

5.2 Virtual Processor Affinity

The Linux kernel's scheduler provides a degree of processor affinity for SMP Linux processes. Processor affinity provides a process that has been made idle a better chance of executing on a CPU with its required environment in cache, once the process becomes active again. As processes, virtual processors are provided the benefits of processor affinity, however this is not very significant, since the bulk of the processing done in the context of a virtual processor is user-level thread processing. Providing for virtual processor affinity, will increase the chances of a user-level thread executing on a CPU that has its environment cached. There are at least two techniques which may be used to provide for virtual processor affinity, one which utilizes the existing global run-queue and one which equips each VP with its own local run-queue.

If virtual processor affinity is to be provided using the existing global run-queue, a scheme similar to that used by the SMP Linux kernel's scheduler may be utilized. [3] When a virtual processor becomes available, a "goodness" value is calculated, factoring in the amount of execution time a threads has had and the virtual processors that provided it. The thread with the highest "goodness" value is the one that is executed. The disadvantage to this approach is that choosing the next thread to execute is no longer a simple task, and the number of operations increases as more threads are added. The current MLTP scheduling implementation does not increase in operations required as threads are added.

If each virtual processor is assigned its own private run-queue, under normal operation, a VP may only run threads that are in its run-queue. In the event that a VP's run-queue has become empty, a mechanism for executing a thread from another VP's run-queue is required; this could become complicated. Scheduling using VP-specific run-queues, provides for a strong virtual processor affinity, and does not increase in the number of operations required as the number of user-level threads increase. However, this method makes no attempt to distribute the user-level threads so that they have an equal opportunity for execution. A thread on one VP which does not yield, prevents all other threads in that VP's queue from being executed.

Regardless of the mechanism used to implement virtual processor affinity, the barrier primitive will have to be redesigned. The current barrier implementation takes advantage of the round-robin scheduling implemented in the global run-queue and places the blocked threads at the end of the queue. Implementing a goodness calculation will cause the run-queue to loose its ordering and allow blocked threads to continuously run and block, before other threads have had the opportunity to enter the barrier. If each processor is given its own run-queue, then the likelihood of all threads entering the barrier once one thread has reached the head of its queue has diminished. MLTP barriers take advantage of that likelihood. Either set of drawbacks may be resolved by using conditional waiting as the blocking mechanism inside a barrier.

5.3 Multiprogramming Coordination

As demonstrated by the ocean benchmark, an application’s performance may degrade when the number of virtual processors exceeds the number of available CPUs. In the presence of multiprogramming, the number of available CPUs is not subject to the number of ready processes in a single application, it is subject to the number of ready processes in all applications. The UCSB implementation of TMPI threads adds another layer of control to the thread package. [15] A daemon processes is added to control the number of processes used by all threaded applications. This processes ensures that the number of virtual processors utilized by all the threaded applications does not exceed the number of CPUs available on the system. It does this by rationing the number of VPs used among all threaded applications. If the total number of ready processes on a system can be maintained at a level that does not exceed the total number of CPUs, situations where VPs running lock-holding threads are forced to idle so that another process may use the CPU, will not occur. It was this situation that degraded the performance of the ocean benchmark using spin locks (section 4.4).

5.4 Dynamic Thread Stack Space Allocation

When threads are created, their entire stack space is allocated from the heap. Though advanced allocation of stack space does little to effect the execution time of an application, it does require apriori knowledge of the maximum amount of stack space a thread will require. MLTP currently fixes this value at 128K bytes. If this limit is exceed a segmentation fault will occur and the application will be aborted. If this space is not used, it will remain in virtual memory, never to be brought into physical memory, however no other threads will be allowed to allocate that space for any other purpose.

It is common among Linux applications to allocate a small stack space and reserve the page immediately above it. If a the allocated space is not enough, a page fault will occur and the faulting application's signal handler will allocate additional stack space. This processes continues to repeat itself until the application has all the stack space that it requires. Implementing a similar scheme for MLTP will allow smaller initial stack space allocations and prevent threads which require more than their initial stack space from causing an application to abort.

5.5 Dynamic Allocation of User-Level Threads

Due to the current algorithm for determining when a VP should be terminated, MLTP does not fully support the allocation of user-level threads after multithreaded processing has begun. It is possible that a VP will terminate prior to the creation of new user-level threads. The scheduling algorithm for a VP may be change, causing the VP to block on a semaphore if there are no available user-level threads. While in the blocked state, very few instruction cycles would be used for the idle VPs. Upon the creation of a user-level thread, or the release of user-level threads from a conditional wait, a signal may be sent to the semaphore which allows the blocked VPs to unblock.

Since VPs will no longer have the ability to terminate themselves, special termination logic must be added to the user layer. Once the last user-level thread has terminated, additional termination logic would terminate all the VPs and free the blocking semaphore. This new termination code would be required in any of the routines capable of terminating a user-level thread.

Chapter 6 Conclusion

MLTP provides a robust and extensible two-layer thread package for SMP Linux. It has been demonstrated that MLTP is suitable for the implementation of a wide variety of thread applications. In general, its performance results are competitive with conventional kernel-level threads. With the addition of further optimizations, it is likely that the performance of MLTP may be improved upon.

The synchronization and context switching mechanisms provided by MLTP are often less costly than those of traditional process-based kernel-level threads. This provides MLTP with the ability to outperform kernel-level threads in applications which require a substantial amount of context switching and synchronization. However, applications that do not require a substantial amount of context switching or synchronization, may suffer a slight penalty from the added overhead of MLTP thread layering.

Appendix A: Timing Data

In this section the timing data for each of the benchmarks in Chapter 4 is presented in spreadsheet format. The graphs presented in Chapter 4 provide an indication of average performance and allow a reader with casual interests to discern information about the relative level of performance. The spreadsheets presented in this section allow readers to study the actual timing results obtained.

The machines that data was collected on are described in Chapter 4. For each of the spreadsheets below, jkthreads were used as the one-to-one (kernel-level) thread that MLTP performance is compared against. Columns labeled "1-to-1" indicated the results obtained when using jkthreads.

A-1 Context Switching

The spreadsheets presented in this section contain the instruction cycle counts obtained by PCL under each of the indicated conditions. The results of the data below are described in section 4.1. The spreadsheet columns indicate the number of context switches, and the cells in the rows are the number of instruction cycles counted for each individual execution. The exception being that the last value for each collection of context switch counts is the average for that column.

Switches	1	2	4	8	16	32
	3153	3652	5022	7763	13194	24068
	2489	3156	4911	7789	13230	24133
	2465	3527	4932	7539	13165	24152
	3032	3534	5046	7689	13308	23785
	2419	3516	4987	7680	13182	24073
	2922	3569	4894	7736	13287	24096
	2505	3512	4886	7747	13244	24104
	2747	3528	4802	7688	13256	24070
	2880	3579	4916	7728	13000	24117
	2859	3524	4884	7613	13160	24061
Average	2747.1	3509.7	4928	7697.2	13202.6	24065.9
Switches	64	128	256	512	1024	
	46038	89568	177100	351692	701916	
	46023	89700	177116	351971	701683	
	45972	89663	177440	351954	701620	
	46078	89695	177152	351940	701770	
	45993	89265	177099	352012	701839	
	46012	89689	177918	351471	701997	
	45986	89652	177124	351935	701728	
	45959	89640	177105	351956	702020	
	45956	89675	177090	352009	701649	
	45514	89582	177048	351983	701628	
Average	45953.1	89612.9	177219.2	351892.3	701785	

Table 1 User-Level Instruction Cycles per MLTP Context Switch

Switches	1	2	4	8	16	32
	4719	3046	3398	3751	5189	6792
	4688	3121	3325	4041	5072	7343
	2886	3110	3365	3871	5254	7500
	2944	2976	3337	4011	5158	7467
	2884	2936	3430	3923	5201	6914
	2903	3027	3517	4043	5275	7500
	2796	3124	3259	4000	4882	7535
	2932	3001	3599	4107	5151	7483
	2936	2972	3399	3751	5147	6897
	2947	3142	3429	4053	5031	7398
Average	3263.5	3045.5	3405.8	3955.1	5136	7282.9
Switches	64	128	256	512	1024	
	12012	18841	39260	66910	148322	
	11976	21132	39454	75563	148246	
	10820	21085	34765	75755	130885	
	12064	21038	39141	75933	148314	
	11943	21106	39348	75650	148448	
	12008	21137	39277	75842	148583	
	10889	21011	34771	75559	131039	
	12038	21119	39148	75579	148524	
	11888	18799	39280	66811	148268	
	12039	21123	39317	75701	148586	
Average	11767.7	20639.1	38376.1	73930.3	144921.5	

Table 2 User-Level Instruction Cycles per Process Context Switch

Switches	1	2	4	8	16	32
	3835	6147	8140	12101	20312	36204
	3484	5546	8069	12107	20313	36221
	4728	5594	8291	11978	20133	36276
	4732	6067	8037	12029	20248	35854
	5119	6124	8128	12127	20293	36297
	4680	6060	8196	12079	20293	36252
	4754	6191	7960	12156	20282	36317
	4857	6060	7862	12216	20257	36325
	5016	6028	8072	12200	19695	36388
	5032	6164	8132	12233	20280	36313
Average	4623.7	5998.1	8088.7	12122.6	20210.6	36244.7
Switches	64	128	256	512	1024	
	68481	132796	267521	518213	1040088	
	68528	132665	261245	518283	1033189	
	68421	132677	261446	524761	1035289	
	68501	132704	261373	518639	1033404	
	68604	132380	263189	518736	1033109	
	68456	132741	261444	518411	1033205	
	68385	132744	261293	518633	1082696	
	68672	132705	261369	520700	1039765	
	68329	132635	261404	518705	1033193	
	68144	132826	261317	518637	1035184	
Average	68452.1	132687.3	262160.1	519371.8	1039912.2	

Table 3 Total Instruction Cycles per MLTP Context Switch

Switches	1	2	4	8	16	32
	8791	10912	14800	23431	39424	71286
	9059	11309	15709	23517	39668	70965
	8838	10824	15402	23100	39156	71238
	8755	10438	15164	23481	39526	71698
	8690	10898	15433	23282	39256	71275
	8796	10792	15352	23495	39871	71244
	8730	10735	15714	23395	39310	71258
	8567	10608	15561	23307	39483	71556
	8860	10866	15262	23459	39181	77946
	8520	10813	15036	23337	39137	71708
Average	8760.6	10819.5	15343.3	23380.4	39401.2	72017.4
Switches	64	128	256	512	1024	
	135033	262892	518325	1030081	2085345	
	135179	264636	522799	1051140	2101907	
	134826	262821	518352	1042975	2087195	
	134885	262648	522408	1057582	2096575	
	134910	269830	518385	1042600	2080019	
	136334	267035	522632	1052569	2079817	
	134947	262777	518390	1049611	2080614	
	136188	264819	524611	1050843	2095971	
	134752	262877	525080	1042675	2094588	
	136148	262812	524263	1058035	2153106	
Average	135320.2	264314.7	521524.5	1047811.1	2095513.7	

Table 4 Total Instruction Cycles per Process Context Switch

A-2 Pi Approximation

The spreadsheets presented in this section contain the wall clock time, measured in seconds, obtained by executing the Pi approximation benchmark under each of the indicated conditions. The results of the data below are described in section 4.2. The spreadsheet columns indicate the number of threads that were used, and the cells in the rows contain the time measured for each individual execution. The exception being that the last value for each collection of times is the average for that column.

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
	9.234281	9.234304	9.237008	9.234469	9.236174	9.233511
	9.233643	9.234541	9.236081	9.238491	9.236863	9.233684
	9.234873	9.233852	9.233972	9.235252	9.235759	9.23356
	9.23457	9.235012	9.23754	9.237893	9.236619	9.235986
	9.233242	9.237406	9.235455	9.234447	9.236818	9.233757
	9.23416	9.233418	9.235519	9.237143	9.235489	9.235474
	9.236152	9.233277	9.234304	9.235565	9.234681	9.233636
	9.234893	9.236569	9.236784	9.236172	9.237859	9.2337
	9.233046	9.233397	9.235067	9.235819	9.236044	9.233978
	9.233866	9.23695	9.238613	9.234802	9.237786	9.233491
Average	9.2342726	9.2348726	9.2360343	9.2360053	9.2364092	9.2340777

Table 5 Pi Approximation 2 Processors 1 Process

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
		4.620377	4.619992	4.620623	4.620464	4.619574
		4.620068	4.620469	4.622873	4.629871	4.622025
		4.619022	4.61956	4.620201	4.620826	4.621819
		4.619182	4.620943	4.620357	4.622377	4.621635
		4.619259	4.619726	4.620018	4.620722	4.622625
		4.622508	4.619649	4.620818	4.620484	4.622627
		4.620146	4.620104	4.619678	4.620921	4.622453
		4.619571	4.621756	4.620691	4.622622	4.624087
		4.620424	4.620259	4.620179	4.621247	4.621569
		4.620053	4.619772	4.62261	4.620176	4.622372
Average		4.620061	4.620223	4.6208048	4.621971	4.6220786

Table 6 Pi Approximation 2 Processors 2 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
			4.622984	4.645559	4.660964	4.620377
			4.639676	4.644335	4.669678	4.64868
			4.648696	4.644773	4.642947	4.634102
			4.644749	4.633382	4.668451	4.667569
			4.641425	4.644261	4.669703	4.630355
			4.628041	4.669589	4.670609	4.645744
			4.642401	4.668419	4.66818	4.623561
			4.621519	4.620966	4.632371	4.643317
			4.663689	4.629917	4.632766	4.644224
			4.6658	4.711024	4.669249	4.640229
Average			4.641898	4.6512225	4.6584918	4.6398158

Table 7 Pi Approximation 2 Processors 4 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
				4.623335	4.657028	4.621843
				4.680071	4.621773	4.643066
				4.652402	4.670024	4.654313
				4.643452	4.693022	4.686457
				4.622029	4.626144	4.626259
				4.647888	4.622136	4.658999
				4.663324	4.642699	4.664481
				4.672676	4.680773	4.639599
				4.671699	4.647949	4.623546
				4.643667	4.633138	4.657586
Average				4.6520543	4.6494686	4.6476149

Table 8 Pi Approximation 2 Processors 8 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
	7.404541	7.402521	7.405319	7.405631	7.404402	7.402586
	7.401668	7.403126	7.402312	7.402882	7.403819	7.402641
	7.402606	7.402034	7.401825	7.402954	7.403453	7.402764
	7.402167	7.403026	7.406672	7.402508	7.406032	7.401687
	7.402919	7.403228	7.402864	7.402852	7.404318	7.405363
	7.404604	7.404117	7.405462	7.402646	7.40363	7.402353
	7.402399	7.402439	7.403287	7.405077	7.404377	7.402177
	7.402464	7.403147	7.402332	7.403226	7.403758	7.405526
	7.401942	7.402901	7.40296	7.403363	7.403885	7.402061
	7.402409	7.402368	7.403192	7.403443	7.403841	7.402146
Average	7.4027719	7.4028907	7.4036225	7.4034582	7.4041515	7.4029304

Table 9 Pi Approximation 4 Processors 1 Process

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
		3.702589	3.7022	3.702534	3.70316	3.701506
		3.702009	3.70175	3.702134	3.702689	3.701508
		3.70401	3.70198	3.701859	3.703182	3.701467
		3.70168	3.702079	3.703057	3.702937	3.701577
		3.702547	3.702009	3.702163	3.703198	3.703874
		3.701449	3.702546	3.702639	3.703101	3.702209
		3.7018	3.70236	3.702141	3.702635	3.701664
		3.701881	3.70205	3.702157	3.702729	3.701527
		3.702663	3.702132	3.702793	3.703387	3.701972
		3.70183	3.701596	3.702311	3.702991	3.703033
Average		3.7022458	3.7020702	3.7023788	3.7030009	3.7020337

Table 10 Pi Approximation 4 Processors 2 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
			1.851889	1.859587	1.862624	1.851426
			1.861868	1.862164	1.859864	2.07015
			1.860253	1.860045	1.862808	1.851503
			1.861663	1.862337	1.859851	1.851683
			1.859965	1.859832	1.862612	1.851464
			1.862436	1.86214	1.860593	1.851628
			1.85995	1.862302	1.86561	1.851305
			1.861974	1.863928	1.866391	1.851336
			1.860053	1.861973	1.862827	1.851602
			1.862211	1.859839	1.859699	1.853923
Average			1.8602262	1.8614147	1.8622879	1.873602

Table 11 Pi Approximation 4 Processors 4 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
				1.888944	1.992082	1.851718
				1.959462	1.955478	1.86663
				1.91331	2.047204	1.866631
				1.967767	2.025842	1.851489
				1.982892	1.99677	1.851766
				1.967183	2.005817	1.863036
				1.872467	2.006479	1.900856
				1.852905	2.006149	1.86137
				1.96947	2.006207	1.854287
				1.980293	1.964185	1.853734
Average				1.9354693	2.0006213	1.8621517

Table 12 Pi Approximation 4 Processors 8 Processes

A-3 Matrix to Matrix Multiplication with Dynamic Scheduling

The spreadsheets presented in this section contain the wall clock time, measured in seconds, obtained by executing the matrix to matrix multiplication benchmark under each of the indicated conditions. The results of the data below are described in section 4.3.1. The spreadsheet columns indicate the number of threads that were used, and the cells in the rows contain the time measured for each individual execution. The exception being that the last value for each collection of times is the average for that column.

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
	2.848213	2.514819	2.535601	2.520509	2.580152
	2.851053	2.513388	2.550807	2.533712	2.559925
	2.853829	2.51539	2.539296	2.521993	2.570979
	2.847966	2.512254	2.545324	2.539441	2.561824
	2.858545	2.520457	2.55287	2.536275	2.548389
	2.845518	2.515132	2.54084	2.527337	2.559485
	2.878955	2.514535	2.57079	2.581137	2.565458
	2.844274	2.516381	2.522867	2.570214	2.561035
	2.850802	2.515715	2.540864	2.566039	2.586618
	2.881666	2.5167	2.54195	2.553456	2.568745
Average	2.8560821	2.5154771	2.5441209	2.5450113	2.566261

Table 13 MLTP 256x256 Matrix Multiplication Dynamic Scheduling 2 Processors

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
	3.087396	2.825495	2.756416	2.745822	2.774554
	3.145318	2.949868	2.789036	2.774888	2.733186
	3.161764	2.828036	2.773025	2.747765	2.740832
	3.149362	2.953078	2.765861	2.754548	2.719412
	3.145687	2.795414	2.777415	2.749135	2.761413
	3.115719	3.034609	2.768676	2.788715	2.742892
	3.123285	2.915487	2.76951	2.767398	2.760342
	3.164327	2.869535	2.767883	2.780127	2.731347
	3.139276	2.878177	2.78907	2.758319	2.780393
	3.105038	3.020123	2.74771	2.800577	2.752732
Average	3.1337172	2.9069822	2.7704602	2.7667294	2.7497103

Table 14 JKThreads 256x256 Matrix Multiplication Dynamic Scheduling 2 Processors

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
	2.448524	2.643282	2.216344	2.189352	2.207003
	2.486067	2.837412	2.207479	2.231609	2.184028
	2.509328	2.745954	2.189094	2.200657	2.22478
	2.362128	2.60887	2.266309	2.189403	2.209541
	2.519986	2.593148	2.219139	2.206182	2.219796
	2.513842	2.671851	2.218205	2.205371	2.208803
	2.506359	2.810885	2.204324	2.22393	2.228547
	2.388052	2.622987	2.25103	2.201759	2.183827
	2.473096	2.605394	2.238231	2.197753	2.231136
	2.463195	2.735509	2.209191	2.206586	2.220494
Average	2.4670577	2.6875292	2.2219346	2.2052602	2.2117955

Table 15 MLTP 256x256 Matrix Multiplication Dynamic Scheduling 4 Processors

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
	2.804395	2.588874	2.121507	2.092133	2.167776
	2.686129	2.641061	2.111362	2.129445	2.117974
	2.774894	2.60847	2.103078	2.14078	2.114307
	2.797741	2.658972	2.132601	2.12518	2.124048
	2.733306	2.592926	2.107258	2.094696	2.185796
	2.722752	2.636517	2.108268	2.136117	2.123858
	2.769754	2.639794	2.110347	2.144937	2.128297
	2.769689	2.677927	2.110387	2.117572	2.112273
	2.740459	2.612199	2.128031	2.088334	2.153238
	2.756944	2.633146	2.107338	2.11518	2.144394
Average	2.7556063	2.6289886	2.1140177	2.1184374	2.1371961

Table 16 JKThreads 256x256 Matrix Multiplication Dynamic Scheduling 4 Processors

A-4 Matrix to Matrix Multiplication with Fixed Scheduling

The spreadsheets presented in this section contain the wall clock time, measured in seconds, obtained by executing the matrix to matrix multiplication benchmark under each of the indicated conditions. The results of the data below are described in section 4.3.2. The spreadsheet columns indicate the number of threads that were used, and the cells in the rows contain the time measured for each individual execution. The exception being that the last value for each collection of times is the average for that column.

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
	2.844307	2.801274	2.766874	2.791082	2.776465	2.860749
	2.843256	2.770061	2.78692	2.789793	2.791691	2.814297
	2.843225	2.76484	2.790005	2.778231	2.773303	2.847724
	2.836136	2.764939	2.797471	2.774257	2.779012	2.866608
	2.83875	2.782163	2.799294	2.762537	2.773318	2.857228
	2.832222	2.757717	2.797831	2.812623	2.776576	2.848552
	2.831414	2.77689	2.801056	2.785366	2.774027	2.836552
	2.828423	2.77337	2.794966	2.786391	2.773798	2.905849
	2.830832	2.776459	2.801039	2.783403	2.772451	2.847194
	2.800891	2.793216	2.797905	2.801558	2.791459	2.842926
Average	2.8329456	2.7760929	2.7933361	2.7865241	2.77821	2.8527679

Table 17 256x256 Matrix Multiplication Fixed Scheduling 2 Processors 1 Process

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
		1.421171	1.416861	1.446822	1.418257	1.462551
		1.423897	1.428213	1.432815	1.436095	1.474039
		1.416882	1.411989	1.446241	1.435432	1.428083
		1.42049	1.420072	1.422248	1.445528	1.414813
		1.419211	1.413676	1.433354	1.436033	1.464333
		1.421955	1.41494	1.423482	1.444791	1.465091
		1.422243	1.443615	1.424449	1.436121	1.438091
		1.418782	1.430965	1.417704	1.44198	1.437761
		1.419244	1.443451	1.418527	1.418482	1.445132
		1.422289	1.434697	1.417755	1.430641	1.466406
Average		1.4206164	1.4258479	1.4283397	1.434336	1.44963

Table 18 256x256 Matrix Multiplication Fixed Scheduling 2 Processors 2 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
			1.423681	1.474336	1.419104	1.440921
			1.421935	1.447358	1.423146	1.433179
			1.41165	1.438516	1.464462	1.500805
			1.423933	1.447047	1.480557	1.453583
			1.419447	1.409877	1.499373	1.487063
			1.424921	1.413287	1.495399	1.470919
			1.437014	1.422679	1.475165	1.484471
			1.432736	1.415008	1.441481	1.497236
			1.408762	1.426866	1.502038	1.505911
			1.424361	1.425913	1.453456	1.447682
Average			1.422844	1.4320887	1.4654181	1.472177

Table 19 256x256 Matrix Multiplication Fixed Scheduling 2 Processors 4 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
				1.4408	1.468584	1.487599
				1.444352	1.447084	1.465058
				1.493135	1.460921	1.471858
				1.452261	1.479167	1.448641
				1.440687	1.500418	1.493282
				1.441137	1.478469	1.45519
				1.469078	1.490543	1.469295
				1.48371	1.441496	1.420201
				1.459404	1.476513	1.495581
				1.441535	1.484452	1.449251
Average				1.4566099	1.4727647	1.4655956

Table 20 256x256 Matrix Multiplication Fixed Scheduling 2 Processors 8 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
					1.535328	1.47206
					1.575794	1.467746
					1.530022	1.423713
					1.533019	1.469717
					1.58732	1.470877
					1.517814	1.442744
					1.534508	1.461423
					1.524691	1.465708
					1.539092	1.430603
					1.587539	1.502235
Average					1.5465127	1.4606826

Table 21 256x256 Matrix Multiplication Fixed Scheduling 2 Processors 16 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
	2.424713	2.477189	2.435789	2.454513	2.465812	2.460909
	2.418067	2.492341	2.427078	2.444123	2.404354	2.482437
	2.45287	2.468225	2.461799	2.416295	2.478388	2.358659
	2.448647	2.364705	2.50096	2.436935	2.505746	2.422223
	2.452369	2.447519	2.34537	2.430538	2.4537	2.407689
	2.368122	2.425231	2.44716	2.413367	2.430777	2.469409
	2.464854	2.43468	2.389698	2.482237	2.403779	2.366347
	2.473118	2.391831	2.440353	2.440142	2.495853	2.43503
	2.424587	2.443806	2.453807	2.466483	2.489775	2.382982
	2.36219	2.442499	2.401482	2.476356	2.452088	2.469222
Average	2.4289537	2.4388026	2.4303496	2.4460989	2.4580272	2.4254907

Table 22 256x256 Matrix Multiplication Fixed Scheduling 4 Processors 1 Process

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
		1.234932	1.225592	1.245331	1.253936	1.222627
		1.308783	1.2259	1.223854	1.217805	1.274139
		1.224318	1.254048	1.267394	1.251575	1.219169
		1.235073	1.219729	1.238615	1.23589	1.253242
		1.192449	1.260207	1.208883	1.223921	1.232279
		1.243967	1.257401	1.252154	1.218482	1.271216
		1.22127	1.22993	1.193181	1.180567	1.217311
		1.231867	1.203631	1.254831	1.23181	1.235743
		1.231189	1.231727	1.25592	1.21779	1.259327
		1.228929	1.242821	1.248786	1.249893	1.248577
Average		1.2352777	1.2350986	1.2388949	1.2281669	1.243363

Table 23 256x256 Matrix Multiplication Fixed Scheduling 4 Processors 2 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
			0.646394	0.645601	0.643106	0.634831
			0.616928	0.84463	0.668785	0.627703
			0.646457	0.664322	0.809526	0.641917
			0.624011	0.721758	0.818471	0.619926
			0.613068	0.638831	0.781305	0.646268
			0.593817	0.817289	0.804113	0.642301
			0.645049	0.842215	0.813752	0.637446
			0.621152	0.664253	0.79422	0.632274
			0.636189	0.673737	0.781354	0.654291
			0.918468	0.852621	0.63502	0.647809
Average			0.6561533	0.7365257	0.7549652	0.6384766

Table 24 256x256 Matrix Multiplication Fixed Scheduling 4 Processors 4 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
				0.725972	0.655485	0.725903
				0.740757	0.674547	0.696268
				0.75893	0.662241	0.639761
				0.733916	0.645799	0.733293
				0.754209	0.675914	0.632718
				0.723649	0.694678	0.63115
				0.699611	0.657048	0.628625
				0.738549	0.657121	0.730928
				0.707937	0.64588	0.710726
				0.641313	0.691496	0.645357
Average				0.7224843	0.6660209	0.6774729

Table 25 256x256 Matrix Multiplication Fixed Scheduling 4 Processors 8 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
					0.682393	0.647949
					0.711991	0.705316
					0.696549	0.630282
					0.667256	0.630036
					0.716172	0.686858
					0.639758	0.730946
					0.720912	0.637604
					0.706493	0.637639
					0.634931	0.684719
					0.711491	0.703515
Average					0.6887946	0.6694864

Table 26 256x256 Matrix Multiplication Fixed Scheduling 4 Processors 16 Processes

A-5 SPLASH-2 Ocean Current Simulation

The spreadsheets presented in this section contain the wall clock time, measured in seconds, obtained by executing the SPLASH-2 ocean current simulation benchmark under each of the indicated conditions. The results of the data below are described in section 4.4. Spreadsheets are provided for executions using both spin locks and semaphore locks. The spreadsheet columns indicate the number of threads that were used, and the cells in the rows contain the time measured in microseconds for each individual execution. The exception being that the last value for each collection of times is the average measured in seconds for that column.

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
	2336281	2195283	2196370	2075851	2157612	2410389
	2308176	2201076	2216181	2078473	2145801	2427427
	2272300	2231106	2195829	2083390	2147975	2449238
	2303630	2214046	2183336	2082734	2165660	2424741
	2283538	2216521	2183122	2092122	2134200	2409901
	2283526	2256148	2203349	2095869	2136017	2449853
	2284281	2211572	2203383	2092273	2147162	2423097
	2277045	2227913	2196623	2102563	2122791	2412895
	2264467	2255770	2194331	2096579	2156063	2427021
	2262693	2199062	2203562	2075792	2155117	2428854
Average	2.2875937	2.2208497	2.1976086	2.0875646	2.1468398	2.4263416

Table 27 258x258 Ocean Simulation using Spin Locks 2 Processors 1 Process

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
		1309295	1342429	1288341	1332747	1384371
		1303014	1355352	1298045	1328519	1395208
		1319031	1330672	1306675	1341001	1395009
		1314763	1339517	1313737	1338986	1405320
		1313418	1335028	1309518	1325042	1394990
		1299852	1330653	1297355	1348323	1372677
		1284537	1357264	1309372	1341476	1382987
		1316462	1347901	1284385	1316834	1393692
		1342353	1337224	1291545	1326453	1392391
		1308307	1343858	1294531	1336666	1382403
Average		1.3111032	1.3419898	1.2993504	1.3336047	1.3899048

Table 28 258x258 Ocean Simulation using Spin Locks 2 Processors 2 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
	2516747	2363443	2306714	2223003	2287957	2737456
	2503372	2358054	2312481	2224819	2264048	2724874
	2500594	2370973	2307636	2221953	2277040	2733258
	2485671	2359741	2304357	2226748	2273482	2730263
	2509395	2369626	2314945	2210141	2271370	2719526
	2493989	2379629	2320868	2210427	2278466	2746319
	2496071	2359055	2301518	2217621	2286150	2708043
	2513284	2369414	2297277	2209674	2281013	2734310
	2494926	2357831	2323475	2219943	2278994	2736964
	2496748	2367738	2311309	2206704	2287170	2741535
Average	2.5010797	2.3655504	2.310058	2.2171033	2.278569	2.7312548

Table 29 258x258 Ocean Simulation using Spin Locks 4 Processors 1 Process

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
		1604261	1608555	1571788	1609968	1860067
		1603140	1628846	1575179	1592830	1824929
		1601482	1611581	1550083	1598027	1843298
		1558599	1608724	1577288	1600956	1827834
		1588771	1627939	1562363	1601795	1839072
		1570403	1614090	1577515	1617462	1821359
		1590602	1612053	1569711	1606380	1762562
		1595249	1600195	1573385	1592914	1812975
		1589438	1601708	1550922	1615414	1800416
		1594400	1609320	1561399	1595854	1808142
Average		1.5896345	1.6123011	1.5669633	1.60316	1.8200654

Table 30 258x258 Ocean Simulation using Spin Locks 4 Processors 2 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
			1216683	1370933	1385626	1477851
			1205736	1361682	1401119	1493901
			1237235	1370458	1371325	1480632
			1253653	1362532	1379497	1473213
			1263129	1359524	1388734	1434147
			1218359	1353990	1380646	1478739
			1230325	1358353	1375898	1468268
			1221497	1369569	1381940	1454221
			1226353	1348914	1375188	1417382
			1224867	1369494	1382476	1439771
Average			1.2297837	1.3625449	1.3822449	1.4618125

Table 31 258x258 Ocean Simulation using Spin Locks 4 Processors 4 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
	2381444	2257236	2264162	2199752	2328248	2410389
	2359562	2247668	2266299	2194510	2327115	2427427
	2397806	2242687	2254755	2191411	2316904	2449238
	2349156	2227362	2271872	2188384	2322330	2424741
	2392400	2277747	2276850	2195669	2320271	2409901
	2383523	2260336	2260198	2197655	2326917	2449853
	2374478	2245882	2271728	2188555	2325115	2423097
	2351006	2235702	2270297	2192564	2328368	2412895
	2393823	2252154	2270941	2194010	2321893	2427021
	2394571	2247292	2257369	2189830	2328515	2428854
Average	2.3777769	2.2494066	2.2664471	2.193234	2.3245676	2.4263416

Table 32 258x258 Ocean Simulation using Semaphore Locks 2 Processors 1 Process

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
		1370133	1404721	1390600	1471328	1384371
		1352815	1381221	1375615	1456715	1395208
		1345533	1396830	1390781	1479277	1395009
		1340356	1377198	1398538	1474702	1405320
		1357467	1415340	1378539	1471158	1394990
		1352476	1406442	1373850	1483673	1372677
		1353624	1417106	1384524	1462374	1382987
		1334438	1407320	1384584	1465348	1393692
		1367826	1395174	1383368	1480835	1392391
		1379303	1391882	1387541	1470665	1382403
Average		1.3553971	1.3993234	1.384794	1.4716075	1.3899048

Table 33 258x258 Ocean Simulation using Semaphore Locks 2 Processors 2 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
			1670410	1553781	1572796	1344822
			1613764	1532432	1600675	1344511
			1665553	1491983	1591458	1355242
			1630599	1506667	1593817	1328827
			1614884	1504768	1598422	1349843
			1642737	1544753	1600752	1338467
			1552127	1527768	1570400	1348688
			1664273	1500581	1614813	1334749
			1600502	1517241	1570460	1328555
			1643248	1506521	1575265	1325161
Average			1.6298097	1.5186495	1.5888858	1.3398865

Table 34 258x258 Ocean Simulation using Semaphore Locks 2 Processors 4 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
				6144335	6339111	1360718
				5830620	6457811	1381066
				5657502	6388465	1367093
				6825867	5954516	1358612
				5656299	7049232	1361781
				6507159	7155225	1380233
				5536367	6521085	1341210
				5052006	6193081	1376072
				6329125	6015801	1374284
				5821111	6537992	1372439
Average				5.9360391	6.4612319	1.3673508

Table 35 258x258 Ocean Simulation using Semaphore Locks 2 Processors 8 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
	2723168	2520374	2451310	2346273	2471619	2737456
	2738913	2480887	2470785	2369763	2469014	2724874
	2737445	2468336	2460295	2348639	2469087	2733258
	2739498	2520569	2476791	2341516	2481800	2730263
	2722208	2495576	2441259	2357089	2462106	2719526
	2723874	2459338	2444704	2362821	2471386	2746319
	2718862	2494895	2459505	2352187	2496233	2708043
	2728543	2487803	2474587	2365648	2454285	2734310
	2741435	2490579	2443534	2358977	2472509	2736964
	2746385	2476403	2439740	2354175	2485537	2741535
Average	2.7320331	2.489476	2.456251	2.3557088	2.4733576	2.7312548

Table 36 258x258 Ocean Simulation using Semaphore Locks 4 Processors 1 Process

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
		1738332	1806978	1783264	1846384	1860067
		1830640	1828818	1760031	1873311	1824929
		1792514	1796507	1766627	1830185	1843298
		1828329	1842695	1769332	1835468	1827834
		1812421	1827784	1786134	1833304	1839072
		1827782	1784499	1781289	1851611	1821359
		1825292	1800701	1778189	1856752	1762562
		1756566	1803980	1775782	1844723	1812975
		1781327	1809771	1768570	1867913	1800416
		1794947	1787301	1752973	1849256	1808142
Average		1.798815	1.8089034	1.7722191	1.8488907	1.8200654

Table 37 258x258 Ocean Simulation using Semaphore Locks 4 Processors 2 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
			1703283	1606751	1690412	1477851
			1777072	1599437	1670526	1493901
			1654420	1618084	1666278	1480632
			1721575	1581853	1687419	1473213
			1900387	1620933	1661840	1434147
			1763967	1580826	1694521	1478739
			1784855	1597193	1683318	1468268
			1743910	1598863	1686127	1454221
			1749486	1601690	1691360	1417382
			1784739	1594790	1683546	1439771
Average			1.7583694	1.600042	1.6815347	1.4618125

Table 38 258x258 Ocean Simulation using Semaphore Locks 4 Processors 4 Processes

	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	1-to-1
				2061752	1847040	1451823
				2090608	1803011	1466180
				1943700	1802978	1462977
				2113236	1828767	1444285
				2012104	1799399	1454526
				2017634	1851449	1463743
				1967176	1836633	1448104
				2033716	1843596	1480527
				2119971	1805305	1463554
				2126778	1831684	1456505
Average				2.0486675	1.8249862	1.4592224

Table 39 258x258 Ocean Simulation using Semaphore Locks 4 Processors 8 Processes

Bibliography

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy, "Scheduler Activations: Effective Kernel Support for User-Level Management of Parallelism", *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 53-79, Feb. 1992

[2] M. Bar, "The Linux Process Model", BYTE.Com Magazine, November 1999, http://www.byte.com/column/BYT19991122S0002

[3] M. Bar, "Process Scheduling in Linux", BYTE.Com Magazine, January 2000, http://www.byte.com/column/BYT19991228S0001

[4] R. Berrendorf, H. Ziegler, "PCL - Performance Counter Library ", Central Institute for Applied Mathematics (ZAM) at the Research Centre Juelich , Germany, Version 1.3, November 1999, http://www.fz-juelich.de/zam/PT/ReDec/SoftTools/PCL/PCL.html

[5] B. Cantrill, "Runtime Performance Analysis of the M-to-N Scheduling Model", Department of Computer Science, Brown University, Providence, Rhode Island, Technical Report, CS-96-16, May, 1996

[6] D. Craig, "An Integrated Kernel- and User-Level Paradigm for Efficient Multiprogramming Support", Computer Science Department ,University of Illinois at Urbana-Champaign, Thesis, 1999

[7] H. Dietz, "Linux Parallel Processing HOWTO", v980105, January 1998, http://yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto.html

[8] D. Keppel, "Tools and Techniques for Building Fast Portable Threads Packages", University of Washington, Technical Report UWCSE 93-05-06

[9] J. Koftinoff, "Simple Kernel threads for linux 2.0 in C and C++", version 1.2, http://www.jdkoftinoff.com/linux3.html

[10] X. Leroy, "LinuxThreads Frequently Asked Questions (and Answers)", Version 0.8, http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html

[11] B. Lim, "Reactive Synchronization Methods for Multiprocessors", Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Thesis, 1995

[12] D. Mentré, "Linux SMP HOWTO", v1.9, January 13, 2000, http://www.phy.duke.edu/brahma/smp-faq/

[13] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, " SunOS Multi-thread Architecture", In *Proceedings of Winter USENIX 1991*, Dallas Texas. 1991

[14] D. Rusling, "The Linux Kernel", REVIEW, Version 0.8-3, 1999, http://www.linuxdoc.org/LDP/tlk/tlk.html

[15] K. Shen, H. Tang, T. Yang, "Adaptive Two-level Thread Management for Fast MPI Execution on Shared Memory Machines", In *Proceedings of ACM/IEEE SC'99*

[16] S. Walton, "Linux Threads Frequently Asked Questions (FAQ)", January 1997, http://www.linuxdoc.org/FAQ/Threads-FAQ/index.html

[17] R. Winiewski, "Scalable Spinlocks for Multiprogrammed Systems", Computer Science Department, University of Rochester, Rochester, New York, Technical Report TR-454

[18] S.C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, "The SPLASH-2 Programs: Characterizations and Methodological Considerations", International Symposium on Computer Architecture, June 1995