Adaptive Two-level Thread Management for Fast MPI Execution on Shared Memory Machines

Kai Shen, Hong Tang, and Tao Yang
Department of Computer Science
University of California
Santa Barbara, CA 93106
{kshen, htang, tyang}@cs.ucsb.edu
HTTP://www.cs.ucsb.edu/research/tmpi

Abstract

This paper addresses performance portability of MPI code on multiprogrammed shared memory machines. Conventional MPI implementations map each MPI node to an OS process, which suffers severe performance degradation in multiprogrammed environments. Our previous work (TMPI) has developed compile/run-time techniques to support threaded MPI execution by mapping each MPI node to a kernel thread. However, kernel threads have context switch cost higher than user-level threads and this leads to longer spinning time requirement during MPI synchronization. This paper presents an adaptive two-level thread scheme for MPI to reduce context switch and synchronization cost. This scheme also exposes thread scheduling information at user-level, which allows us to design an adaptive event waiting strategy to minimize CPU spinning and exploit cache affinity. Our experiments show that the MPI system based on the proposed techniques has great performance advantages over the previous version of TMPI and the SGI MPI implementation in multiprogrammed environments. The improvement ratio can reach as much as 161% or even more depending on the degree of multiprogramming.

1 Introduction

Recently shared-memory machines (SMMs) have become popular for high-end computing because of their success in the commercial market. Even though MPI [2, 12, 26] is designed for distributed memory programming, it is important to address performance portability of MPI code on shared memory machines (SMMs). There are three reasons that people use MPI on SMMs. First of all, MPI has been widely used for many parallel applications on large distributed memory machines and clusters of PCs or SMMs and will continue to be popular (or dominating) in the foreseeable future. It is necessary to support popular MPI code on SMMs. Secondly, MPI code on SMMs has better performance portability on other platforms, compared to other programming models such as threads and OpenMP. This is especially important for future computing infrastructures such as information power grids [1, 7, 11], where resource availability, including platforms, dynamically changes for submitted jobs. Finally, even though it is easier to write a thread or OpenMP based parallel program, it is hard for an average programmer to fully exploit the underlying SMM architecture without careful consideration of data placement and synchronization protocols. On the other hand, performance tuning for SPMD-based MPI code on large SMMs is relatively easier since partitioned code without using shared space exhibits good data locality.

Large-scale SMMs are normally multiprogrammed for achieving high throughput. It has been widely acknowledged in the OS community that space/time sharing is preferable, outperforming other alternatives such as pure co-scheduling or gang-scheduling for better job response times [8, 18, 29, 32, 33]. The modern operating systems such as Solaris 2.6 and IRIX 6.5 have adopted such a policy in parallel job scheduling (see a discussion in Section 6 on gang-scheduling used in the earlier version of IRIX). In such environments, the number of processors allocated to an MPI application may dynamically change and it may be less than the number of MPI nodes (or sometime called MPI processes in the literature). On the other hand, MPI uses the process concept and global variables in an SPMD program are non-sharable among MPI nodes. As a result, a conventional MPI implementation that uses heavy-weighted processes for code execution suffers severe performance degradation on multiprogrammed SMMs. To improve MPI code performance, our previous work [27] has developed compile-time and run-time techniques to support threaded MPI execution and demonstrated our optimization techniques for executing a large class of C programs using MPI standard 1.1. The compile-time transformation adopts the thread-specific data mechanism to eliminate the use of global and static variables in C code. The run-time support includes an efficient point-to-point communication protocol based on a novel lock-free queue management scheme. Our experiments on an SGI Origin 2000 show that our MPI prototype called TMPI using the proposed techniques is competitive with SGI's native MPI implementation and MPICH [12] in a dedicated environment, and it has significant performance advantages in multiprogrammed environments.

However, the previous version of TMPI is built upon kernel threads which are directly scheduled by multiprocessor OS and have high context switch cost compared to user-level threads. In a multiprogrammed environment with heavy loads, threads yield their execution during communication synchronization and thus context switches may happen frequently. This consideration leads us to map each MPI node to a user-level thread and then adaptively schedule user-level threads on top of kernel threads. In this paper, we propose techniques to build an adaptive two-level thread system under TMPI. The number of user-level threads is the same as the number of MPI nodes and the number of kernel threads that execute user-level threads is adaptive to the multiprogramming level in the system. The deployment of user-level threads also minimizes wasteful CPU spinning in synchronization. Spinning is necessary to implement inter-node MPI communication but blindly waiting for certain events costs a substantial amount of CPU time. By taking advantage of low context switch cost of user-level threads and exploiting scheduling information exposed from this two-level thread mechanism, we develop an adaptive event waiting strategy to minimize unnecessary spinning overhead and exploit cache affinity.

The rest of this paper is organized as follows. Section 2 gives an overview of TMPI system that serves as the foundation for this work. Section 3 describes our two-level thread management. Section 4 proposes a scheduler-conscious event waiting strategy for efficient MPI communication. Section 5 presents the experimental results on a 4-processor SGI Power Challenge and a 32-processor SGI Origin 2000. Section 6 discusses the related work and concludes the paper.

2 An overview of TMPI

TMPI contains two parts, as shown in Figure 1. The compile-time pre-processing part provides code transformation that allows each MPI node to be executed as a thread. This is accomplished by eliminating global and static variables in the original C program. In an MPI program, each node can keep a copy of its own *permanent variables* – variables allocated statically in the heap, such as global variables and local static variables. If such a program is executed by multiple threads without any transformation, then all threads will access the same copy of permanent variables. To preserve the semantics of an MPI program, it is necessary to make a "private" copy of each permanent variable for each thread. We have developed a preprocessor for C programs based on *thread-specific data (TSD)*, a mechanism available from most thread systems. We implement a TSD scheme for SGI SPROC threads and use the dynamic heap to allocate space for TSD. With TSD, different threads can associate different data with a common key. Given the same key value, each thread can store/retrieve its own copy of data. The details are in [27]. The transformation involves indirect access of TSD-based permanent variables. The overhead of such indirection is insignificant in practice. Thread safety is also addressed in [2, 23], however, their concern is how multiple threads can be invoked in each MPI node (process), but not executing each MPI node as a thread. Currently we assume that each MPI node does not spawn multiple threads and we plan to relax this constraint in the future[1].

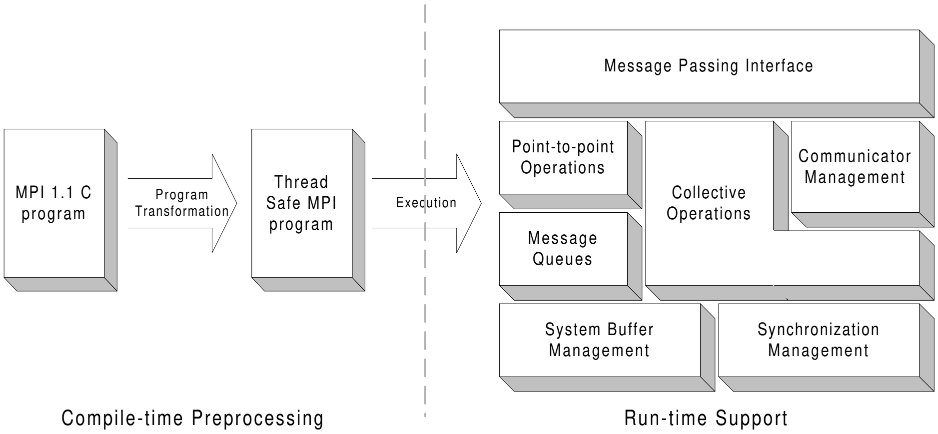

Figure 1: System architecture of TMPI.

The run-time component of TMPI roughly contains three layers. The lowest layer provides support for several common facilities such as buffer and synchronization management, the middle layer is the implementation of various basic communication primitives and the top layer translates the MPI interface to the internal format. The intrinsic difference between the thread model and the process model has a big impact on the design of run-time support. An obvious advantage of multi-threaded execution is the low context switch cost. Besides, inter-thread communication can be made faster by directly accessing threads' buffers between a sender and a receiver, with simplified buffer management. A process-based implementation may allocate a shared segment to exchange information, and its size is usually restricted. Advanced

[1]TSD does not prevent us from relaxing this constraint since we can allocate data area specific to each MPI node. Threads spawned within each MPI node can share data through such an area.

OS features may be used to share a large address space among processes; however, such an implementation is problematic. For example, in an earlier version of the SGI MPI implementation, address-remapping is used to achieve process-space sharing; however, sharing is dropped in the current version because of insufficient address space and software incompatibility [24]. As a result, process-based implementation requires that inter-process communication must go through an intermediate system buffer. Thus a thread-based run-time system can potentially save some memory copy operations. The thread model also gives us flexibility in designing a lock-free communication protocol to further expedite the message passing speed since if a thread holds a lock while being preempted the entire program execution may be delayed substantially [13].

3 Two-level Thread Management

In a multiprogrammed execution environment, the number of processors allocated by an OS to an MPI application may be less than the number of MPI nodes because multiple jobs share the processors and threads may yield their execution during communication synchronization. In that sense, context switches among MPI nodes are unavoidable and they may happen frequently when multiple applications share the machine.

Our earlier work on TMPI [27] maps each MPI node to a kernel thread. However, context switches among kernel threads are expensive because such a context switch requires a kernel trap and at least two stack switches (one switch from old thread stack to the kernel stack, and another switch from the kernel stack to new thread stack). On the other hand, a context switch between two user-level threads is much cheaper. This leads us to map each MPI node to a user-level thread and schedule these user-level threads on top of kernel threads. The number of kernel threads is controlled close to the number of allocated physical processors. This two-level thread scheme minimizes context switches at kernel-level. Figure 2 shows the layers of our new MPI runtime support with two-level thread management.

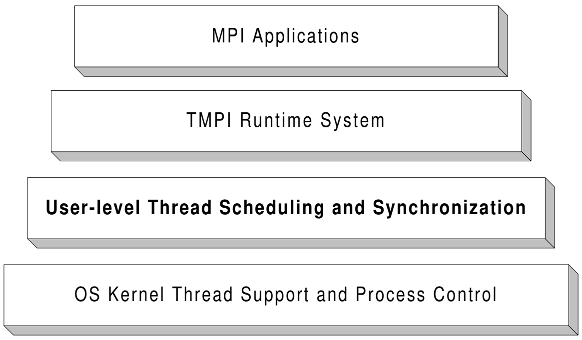

Figure 2: Architecture of our MPI runtime system with two level thread management.

In order to control the number of kernel threads (not exceeding the number of underlying processors), we need to obtain dynamic scheduling and load information in the system. There are two ways to address this issue. 1) OS can be modified slightly to export such information to users. 2) For a class of applications such as MPI programs, a user-level central monitor can collect job request information and decide how to partition processors among applications. The central monitor then places the resource allocation information on a shared memory page that can be read by all the applications. The previous work of Tucker and Gupta [29] controls the total number of processes for parallel applications written in a task-queuing and work-stealing programming model. Yue and Lilja conduct a similar work for fine-grained loop parallelization [31]. In comparison, our research is focused on the trade-off between user and kernel threads for MPI.

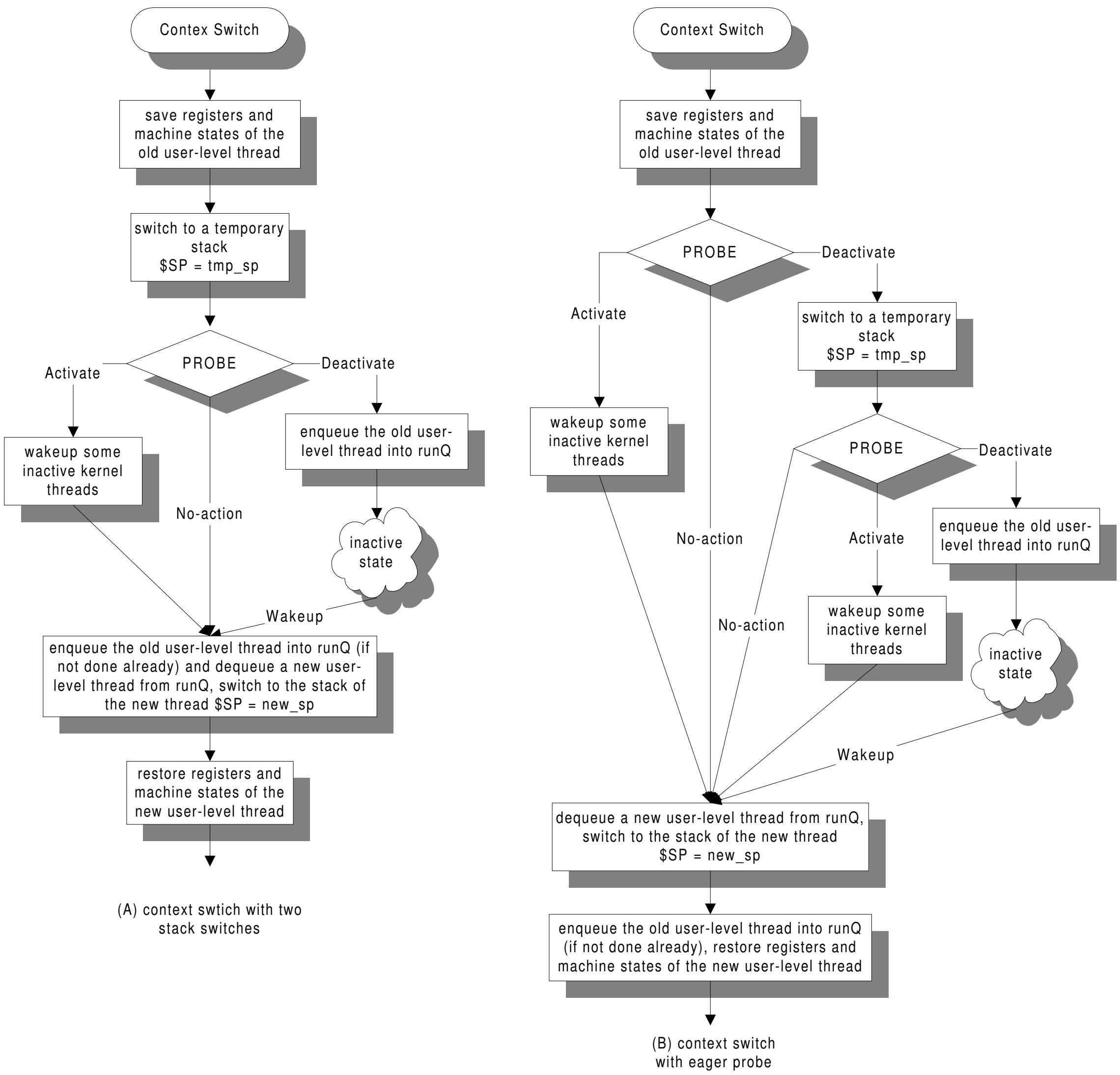

Figure 3: (A) Execution flow of a context switch with two stack switches. (B) Execution flow of a context switch with eager probe.

In our scheme, the system maintains a queue of user-level ready threads (called `runQ` in this paper) and each user-level thread runs as an MPI node. The system also maintains a set of kernel threads and each kernel thread is either active in executing a user-level thread, or inactive. During program execution, a kernel thread picks up a user-level thread from `runQ`, executes it until this program calls an MPI communication primitive which leads to a context switch. During the context switch, this kernel thread calls probing to get resource allocation information and the result of this probing leads to three types of actions:

1. `Deactivate` which means the calling kernel thread should release the user-level thread it executes and suspend itself because the host application has more active kernel threads than what it should have.

2. `Activate` which means the calling kernel thread should wake up some inactive kernel threads because the host application has less active kernel threads than what it should have.

3. `No-action` which means the number of active kernel threads fits the system load and no adjustment is needed.

Because the user-level scheduler cannot interrupt an MPI program during computation, probing can only be invoked when the user program calls an MPI routine. On the other hand, since kernel thread deactivation due to probing is not cheap and scheduling information dynamically changes, excessive probing and adjusting the number of kernel threads does not pay off. In our scheme, we conduct such probing during each user-level context switch.

The result of probing can cause the suspension of a kernel thread. In order to safely release the user-level thread this kernel thread executes in case of suspension, two stack switches are necessary to accomplish a context switch between two user-level threads. The first one switches the context from the old user-level thread stack to a temporary stack. Then the kernel thread can be safely suspended to the inactive state because it does not hold any user-level thread. A second stack switch from the temporary stack to the new user-level thread stack completes the context switch. The detailed flow of such a context switch with two stack switches is illustrated in Figure 3(A).

We propose a technique called *eager probe* to avoid unnecessary stack switches. The execution flow of a context switch with eager probe is illustrated in Figure 3(B). This method can save one stack switch in most cases. It requires the kernel thread conduct the probe in the beginning of a context switch. If the result of this probe is not "`Deactivate`", then we can proceed the context switch with only one stack switch directly from the old user-level thread to a new user-level thread. And the context switch with two stack switches is only used when the first probe returns "`Deactivate`", which happens infrequently. The second probing is needed for this case since the result of the first probing may change after switching to the temporary stack. Note that as show in Figure 3(A), one advantage of the context switch with two stack switches is that the two `runQ` operations (the enqueue of the old thread and the dequeue of the new thread) can happen together when the execution context is in the temporary stack. But for the context switch with one stack switch, these two operations have to split, which incurs more `runQ` management overhead. However, the experiments show that this cost is insignificant comparing to the saving of one context switch. We conduct experiments on an SGI Power Challenge to study the context switch cost of different scheduling schemes and the results are listed in Table 1. We can see that replacing kernel thread with user-level thread brings the context switch cost down from $12\mu s$ to $3\mu s$. And the eager probe brings the cost further down to $2\mu s$.

4 Scheduler-conscious Event Waiting

MPI supports several dozens of communication and synchronization primitives [2, 12, 26]. All these primitives share one common need - a mechanism to wait for asynchronous events. For example, function

Kernel-level context switch	$12\mu s$
User-level context switch with two stack switches	$3\mu s$
User-level context switch with eager probe	$2\mu s$

Table 1: Context switch cost.

MPI_Recv() needs to wait until a desired message is copied to the user buffer.

Our waiting scheme (called `waitEvent`) is illustrated in Figure 4. In this example, there are one *waiter* and one *caller*. The waiter is the thread that needs to wait for certain asynchronous event, and the caller is the thread which triggers the event. It should be noted that there may be multiple waiters or callers in some synchronization primitives such as MPI broadcast, reduce and barrier. In Figure 4, the waiter issues "`waitEvent (*pflag == value)`" to wait until the content pointed by `pflag` is equal to `value`. The caller issues an assignment "`*pflag = value`", which asynchronously wakes up the waiter.

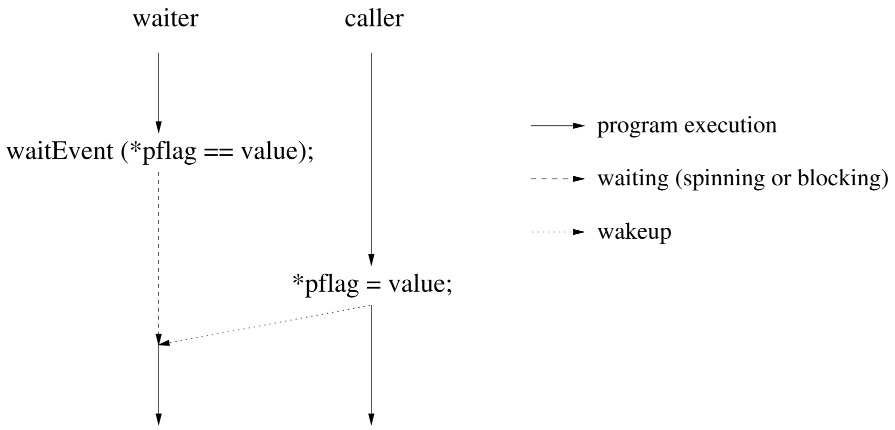

Figure 4: Illustration of a `waitEvent` synchronization.

The waiter thread can either spin or block when waiting for certain event. It has been widely believed that spin-then-block [19], which spins for some time and then blocks, is the right choice to handle synchronization in multiprogrammed environments. Deciding the optimal spinning time before blocking, however, has been the focus of extensive research under various application domains [15, 16, 19]. In particular, it has been shown [16] that the spin-then-block strategy, where the time spent spinning is equal to the blocking cost (including thread suspension, context switch, and thread resumption), is *competitive* to the optimum. Namely, the cost of this strategy, amortized over all attempts, is at most twice that of the optimal off-line algorithm [2].

In our kernel thread based TMPI implementation, instead of fixed spin-block we have used an adaptive spinning-block approach which does linear or exponential backup if the previous spinning is unsuccessful (similar to the random walk algorithm suggested in [15]). However, even though the pure context switch cost of kernel threads is not so expensive in SGI machines, thread yielding and resumption is cumbersome and fairly slow (e.g. the thread yield function resumes a kernel thread in a non-deterministic manner and the shortest sleep interval for a nanosleep call on Power Challenge is $1.6ms$). As a result, the spin time in

[2]An off-line algorithm has complete knowledge about when the caller will execute the wakeup event.

the TMPI event waiting function is fairly large. Using the user-level threads instead of pure kernel threads significantly reduces the blocking cost; hence the spinning period can be shortened proportionally and CPU waste can be minimized in TMPI-2 compared to the case in TMPI.

The above spin-block approach only considers the penalty of blocking in terms of context switch and thread resumption. There is actually more overhead in spinning and blocking: 1) spinning may be useless if the caller thread that executes the wakeup event is not currently being scheduled; 2) blocking may suffer cache refresh penalty due to context switch. The previous work on scheduler-conscious synchronization [4, 17] has considered using OS scheduling information to guide lock and barrier implementations. There is also work on OS scheduling to exploit cache affinity [30]. We combine these two ideas together and extend them for the MPI runtime system. The unique aspect of our situation is that scheduling information is exposed at user-level because of our two-level thread management, and the context switch cost in our system is relatively small (only $2\mu s$ in SGI Power Challenge). On the other hand, the cache penalty is costly for MPI programs. Switching from one MPI node to another MPI node loses cache affinity because MPI nodes do not share data. In comparison, the previous affinity scheduling study [30] considers that switching among user threads from the same application still keeps cache affinities because threads inside one application share global data for general shared memory programming.

Our adaptive waiting technique contains two parts and the execution flow is illustrated in Figure 5:

1. In the beginning of `waitEvent`, the waiter checks if the caller thread is currently scheduled or not. If the caller is not scheduled, spinning doesn't make any sense because the caller is not expected to execute the wakeup event any time soon. So the waiter blocks (yields) immediately without spinning in this case.

2. Since the time spent on executing instructions for context switch is small in our system, the spin time is solely leveraged by the cache penalty. If the user-level thread does not have affinity to the underlying kernel thread, then the spinning makes no sense. Our second strategy is to conduct a *cache affinity* check before starting spinning. This is done by comparing the current underlying kernel thread with the last kernel thread it ran on before calling `waitEvent`. It starts spinning only if the cache affinity check returns true.

We should point out that scheduling a user-level thread on the same kernel thread does not necessarily guarantee the cache affinity because a kernel thread could be deactivated and migrated to different physical processor. However, the frequency of such a migration is normally much lower than that of user-level context switches. Therefore our method can ensure cache affinity in most cases.

It is non-trivial to find the optimal spin time T before blocking. Generally speaking, short spin time reduces wasteful spin time in multiprogrammed environments but long spin time yields optimal performance for dedicated environments and also reduces the contention on the scheduler. To determine the optimal spin time T before blocking, we also need to find the precise cache refresh penalty. This is difficult because this penalty is application and architecture dependent. We have calculated an empirically optimal number of T on our testing benchmarks. In our experiments, we use $T = 50\mu s$.

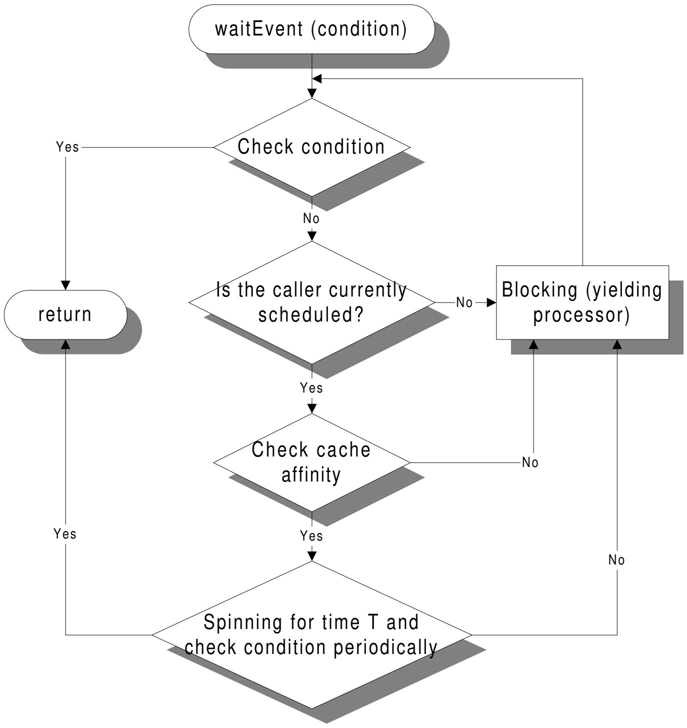

Figure 5: Execution flow of `waitEvent` with *scheduler-conscious event waiting*.

5 Experimental Studies

The purpose of the experiments is to study if the two-level thread management together with scheduler-conscious event waiting can gain great performance advantages in multiprogrammed environments and be competitive to other MPI systems in dedicated environments. By "dedicated", we mean that the load of a machine is light and an MPI job can run on requested number of processors throughout the computation. Being competitive in dedicated situations is important since a machine may swing dynamically between multiprogrammed and dedicated states. All experiments are conducted on a 4-processor SGI Power Challenge multiprocessor (4 200MHz R4400 processors and 256MB memory; each R4400 has 32KB level-1 cache and 4MB secondary cache) and a 32-processor SGI Origin 2000 (32 195MHz R10000 processors and 7860MB memory; each R10000 has 64KB level-1 cache and 4MB secondary cache).

In the following experimental studies, we use symbol TMPI to denote the original TMPI implementation built directly upon kernel threads. And we use TMPI-2 to denote the new TMPI with two-level thread management. Under space/time sharing [18, 29, 33], the MPI nodes from each application are multiplexed into less number of physical processors in the presence of multiprogramming. We define the *multiplexing degree* to be the average number of MPI nodes on each processor for a multiprogrammed environment. Note that the multiplexing degree of each application reflects the multiprogramming level of the whole system.

The characteristics of the test benchmarks are listed in Table 2. Two of them are kernel benchmarks for dense matrix multiplication using Canon's method (MM) and a linear equation solver using Gaussian Elimination (GE). SWEEP3D is from the ASCI application benchmark collection from Lawrence Livermore and Los Alamos National Labs. The last three benchmarks (GOODWIN, TID and E40R0100) are parallel sparse LU factorizations with pivoting [14, 25] applied to three input matrices from circuits simulation and fluid dynamics and the execution of this code involves irregular data-dependent fine-grained communication. We list the synchronization frequency of the test benchmarks in column 3 of Table 2. The synchronization frequency is obtained by running each benchmark with 4 MPI nodes on the 4-processor Power Challenge. And the frequency is expected to be larger when the number of MPI nodes increases. We observe that the three sparse LU benchmarks have much more frequent synchronization than others do.

Benchmark	Function	Synchronization frequency	MPI operations
MM	Matrix multiplication	2 times per second	mostly `MPI_Bsend`
GE	Gaussian Elimination	59 times per second	mostly `MPI_Bcast`
SWEEP3D	3D Neutron transport	32 times per second	mixed, mostly send/recv
GOODWIN	Sparse LU factorization	2392 times per second	mixed
TID	Sparse LU factorization	2067 times per second	mixed
E40R0100	Sparse LU factorization	1635 times per second	mixed

Table 2: Characteristics of the tested benchmarks.

The rest of this section is organized as follows. We will first evaluate and compare TMPI-2 with TMPI and SGI MPI for a multiprogrammed workload containing a sequence of parallel jobs. Then we study impact of multiprogramming in details using workloads with fixed multiplexing degrees. Finally we report experiments which isolate and demonstrate impact of adaptive two-level thread management and scheduler-conscious event waiting.

5.1 Performance evaluation on a multiprogrammed workload

We first report an experiment that uses a multiprogrammed workload which contains multiple jobs and we measure the turn-around time of each job (the elapsed time from the job submission to its completion, including data I/O and initialization). We run this workload on a dedicated machine with different arrival intervals. The dedicated machine we use is the 4-CPU SGI Power Challenge. The workload contains six jobs whose names and submission order are listed below: GOODWIN (sparse LU factorization with 4 MPI nodes), MM2 (MM with 1152x1152 matrix, 4 MPI nodes), GE1 (GE with 1728x1728 matrix, 4 MPI nodes), GE2 (GE with 1152x1152 matrix, 2 MPI nodes), MM1 (MM with 1440x1440 matrix, 4 MPI nodes), and SWEEP3D (4 MPI nodes). For each MPI implementation, we launch these six jobs consecutively with a fixed arrival interval.

Table 3 lists the turn-around time of each job when the launching interval is 20, 14, 12 and 10 seconds respectively. The last row in Table 3 shows the normalized turn-around time for each job, which is calculated by dividing the absolute time by the time of using TMPI-2. The result shows that TMPI-2 is up to 30% faster than TMPI and up to 161% faster than SGI MPI with different launch intervals. When the launch interval decreases, TMPI-2 tends to obtain better improvement ratios. This is because the shorter is the interval, the higher level of multiprogramming is the workload, and the more advantages our adaptive thread management has.

Jobs	interval=20			interval=14			interval=12			interval=10		
	TMPI-2	TMPI	SGI	TMPI-2	TMPI	SGI	TMPI-2	TMPI	SGI	TMPI-2	TMPI	SGI
GOODWIN	16.0	20.4	19.0	18.5	26.2	21.1	20.3	26.3	18.7	20.8	30.5	29.2
MM2	14.9	16.2	25.9	21.5	26.0	42.1	21.8	34.3	43.3	29.3	43.7	60.6
GE1	19.9	21.2	27.3	28.6	34.7	59.0	34.1	47.7	102.1	37.5	65.3	162.0
GE2	11.0	11.8	16.4	12.4	17.8	65.1	11.2	26.0	33.5	22.4	40.5	61.7
MM1	33.8	35.1	63.8	36.5	40.7	122.0	53.3	54.8	122.4	66.5	63.3	160.4
SWEEP3D	47.5	47.4	67.4	54.5	56.2	123.2	59.0	64.2	130.0	67.0	72.9	162.1
Average	23.8	25.4	36.6	28.7	33.6	72.1	33.3	42.2	75.0	40.6	52.7	106.0
Normalized	1.00	1.07	1.54	1.00	1.17	2.51	1.00	1.27	2.25	1.00	1.30	2.61

Table 3: The turn-around time of each job in a workload with different launching intervals on an SGI Power Challenge. SGI stands for SGI MPI. All times are in seconds.

Different launching intervals among consecutive jobs represent different degree of job execution overlap; however, we are not able to know the average degree of job multiplexing. In the next subsection, we use workloads with fixed multiplexing degrees.

5.2 Performance impact of multiplexing degrees

In this experiment, each multiprogrammed workload is designed such that the multiplexing degree is predictable and is kept to be a constant during the entire execution. Each workload only has one benchmark program mentioned above; however each physical processor runs multiple MPI nodes. With multiplexing degree t, every processor runs t MPI nodes and there are total $t * p$ MPI nodes for p processors. This design simulates a multiprogramming situation and also gives a uniform multiplexing degree among processors. In this way, we can clearly identify the impact of multiplexing degrees on performance of TMPI-2, TMPI and SGI MPI. We use the MEGAFLOPS rating or speedups reported by each benchmark. These numbers collected may exclude the time for initial I/O and data distribution.

Experiments on an SGI Power Challenge multiprocessor. Figure 6 depicts the performance of SGI MPI, TMPI and TMPI-2 when running a different number of MPI nodes, which simulates both dedicated and multiprogrammed scenarios. For the performance in dedicated environments (i.e. performance of executing 1-4 MPI nodes), we observe a similar performance among these three implementations across all six benchmarks except MM[3]. This is expected because context switch happens very rarely, if at all, in dedicated environments. However, in multiprogrammed situations for executing 6-16 MPI nodes, TMPI-2 exhibits substantial performance improvement over TMPI and SGI MPI. For the first three computation-intensive benchmarks (MM, GE and SWEEP3D), the performance curve of TMPI-2 does not degrade when the multiplexing degree increases. In certain cases, we even observe the performance in a multiprogrammed environment is better than the performance in a dedicated environment, which is due to the better load balancing in the multiprogrammed situation. And in general for the first three benchmarks, TMPI-2 is more than 50% faster than TMPI and up to 10-fold faster than SGI MPI in an environment with a multiplexing degree of 3. For the last three benchmarks (GOODWIN, TID and E40R0100) which have much

[3]SGI MPI suffers certain performance loss for 4 MPI nodes on MM due to message buffer overflow and we study this issue in details in [27].

more fine-grained synchronization, TMPI-2 does exhibit some performance loss when the multiplexing degree increases. This is foreseeable because fine-grained synchronization causes frequent context switch that brings high overhead and poor cache utilization. However, the other two implementations, SGI MPI and TMPI suffer far greater performance loss on these benchmarks and they are slower than TMPI-2 in several orders of magnitude. We don't have the experimental results for TMPI and SGI MPI for more than 8 MPI nodes because they take too much time to complete.

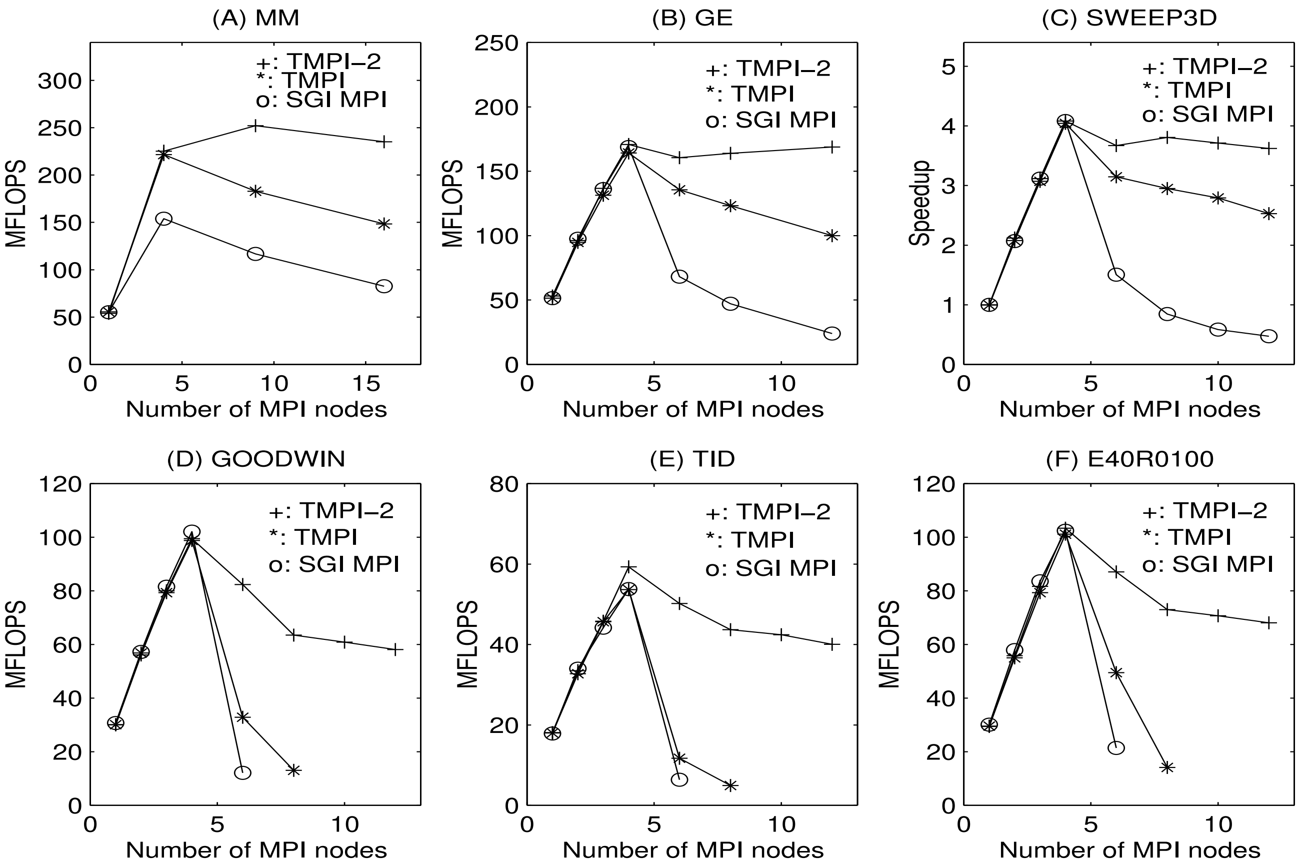

Figure 6: Impact of multiplexing degree on a 4-processor SGI Power Challenge. The purpose of executing MPI nodes on less number of processors is to emulate a multiprogramming workload.

Experiments on an SGI Origin 2000. We also conduct the experiments on a larger machine to compare the performance of TMPI-2, TMPI and SGI MPI under different multiplexing degrees. Table 4 lists the performance ratio of TMPI-2 to TMPI for GE and SWEEP3D, which is the MEGAFLOPS or speedup number of the TMPI-2 code divided by that of the TMPI. Table 5 lists the performance ratio of TMPI-2 to SGI MPI. The improvements on other benchmarks are even larger. We do not provide complete experimental results for other benchmarks because the experiments for TMPI and SGI MPI take too much time to complete.

From Tables 4 and 5, we observe that the performance ratios stay around 1 when the multiplexing degree is 1, which indicates that all three implementations have similar performance in dedicated environments. When the multiplexing (node-per-processor ratio) increases to 2 or 3, TMPI-2 can be up to 88% faster than

Benchmarks	GE			SWEEP3D		
Multiplexing degree	1	2	3	1	2	3
2 processors	1.04	1.11	1.23	1.04	1.08	1.17
4 processors	1.01	1.12	1.45	1.03	1.19	1.36
6 processors	0.99	1.22	1.54	1.01	1.44	1.51
8 processors	0.99	1.32	1.67	1.01	1.63	1.88

Table 4: Performance ratio of TMPI-2 to TMPI on Origin 2000. Multiple MPI nodes are multiplexed to 2-8 dedicated processors, which emulates multiprogramming workloads.

Benchmarks	GE			SWEEP3D		
Multiplexing degree	1	2	3	1	2	3
2 processors	1.01	3.05	7.22	1.01	2.01	2.97
4 processors	1.02	5.10	13.66	1.00	3.71	7.07
6 processors	1.03	6.61	22.72	1.00	4.44	11.94
8 processors	1.03	8.07	32.17	1.00	6.50	15.69

Table 5: Performance ratios of TMPI-2 to SGI MPI.

TMPI and up to 32-fold faster than SGI MPI. We also observe that the performance improvement increases when the number of physical processors or the multiplexing degree increases. This performance gain over TMPI is due to user-level threads' low context switch cost and our scheduler-conscious strategy that exploits cache affinity and minimizes unnecessary spinning. Our system outperforms SGI MPI substantially because SGI MPI is process-based and has high overhead in their synchronization and communication primitives.

5.3 Benefits of optimization techniques

Benefits of user-level threads in fast synchronization. Because the high context switch cost and high blocking/unblocking overhead of kernel threads, the spin time in the TMPI's event waiting function before processor yielding is quite large (up to $2ms$ on Power Challenge). For TMPI-2, the low context switch cost and the negligible blocking/unblocking overhead make it possible to use a small spin time ($50\mu s$) without sacrificing performance in dedicated environments. Table 6 shows the performance improvement of TMPI-2 with $50\mu s$ spin time over TMPI-2 with $2ms$ spin time on our SGI Power Challenge machine. In dedicated environments (when the multiplexing degree is 1), we observe that TMPI-2 with $50\mu s$ spin time is no more than 2% slower than TMPI-2 with $2ms$ spin time. But in multiprogrammed environments, TMPI-2 with short spin time performs much better (up to 73% faster than TMPI-2 with long spin time). Notice that blocking cost in TMPI-2 is much lower than TMPI, thus there is a large performance difference between TMP-2 and TMPI even with the same spin time. This experiment only illustrates the impact of short spin time.

It should be noted that in this experiment, we fix 4 processors and increase the number of MPI nodes. This scaling also affects the total number of synchronization among MPI nodes since more communication

# of MPI node	4	8	12
Multiplexing degree	1	2	3
GE	-1.9%	1.2%	3.3%
SWEEP3D	-0.2%	1.6%	7.3%
GOODWIN	-1.2%	18.5%	61.8%
TID	-0.9%	17.5%	73.2%
E40R0100	-2.0%	19.1%	59.6%

Table 6: Performance improvement ratio of TMPI-2 with $50\mu s$ spin time over TMPI-2 with $2ms$ spin time on an SGI Power Challenge.

synchronization is involved when more MPI nodes are deployed. Table 7 shows the total number of synchronization and blocking during the execution of GOODWIN with TMPI-2 and $2ms$ spin time on this SGI Power Challenge and the number of synchronization increases as the number of MPI nodes increases. However this table also indicates that there are more context switches due to blocking when the multiplexing degree increases, by examining the ratio of blocking over synchronization listed in the last column. Notice that during a communication synchronization, a `waitEvent` routine is called as discussed in Section 4, the system spins and may or may not enter the blocking state. In a dedicated environment with multiplexing degree 1, there is only 3.0% of synchronizations leading to blocking. This number climbs to 25.7% in a multiprogrammed environment with a multiplexing degree of 3. Thus if we were able to run a job with 4 MPI nodes on one physical processor on this machine, we would also expect a high blocking ratio.

The fourth column of Table 7 also lists the number of blocking per second for each multiprogramming case. With a multiplexing degree of 3, the rate is 407 times/sec. Since spinning is conducted before each time entering blocking state and blocking in TMPI-2 is very fast, for TMPI-2 with $50\mu s$ spin time, on average there is a $5ms$ (0.00005*407/4) synchronization delay on each processor during each elapsed second, which is insignificant. For TMPI, each blocking action could cost up to $2ms$ and each spinning action costs up to $2ms$. The synchronization delay could be about 0.4 seconds (0.004*407/4) on each processor during each elapsed second. Namely over 40% of the total elapsed time could be caused by synchronization when TMPI executes this MPI program (sparse LU for matrix GOODWIN). Notice that the actual synchronization delay could be more since this calculation does not include successful spins with no blocking involved. Thus we can conclude that the improvement of TMPI-2 over other implementations in multiprogrammed environments is largely due to the saving in context switch and spinning.

Benefits of scheduler-conscious event waiting. We conducted experiments to study benefits of scheduler-conscious event waiting. Table 8 and Table 9 show the performance improvement of scheduler-conscious event waiting over the basic spin-then-block method on the Power Challenge and the Origin 2000 respectively. All experiments are conducted on 4 physical processors. We exclude MM because the code itself has a restriction that the number of MPI nodes needs to be the square number of an integer. We have two observations from the data listed in Table 8 and Table 9. First, the improvement is more substantial for benchmarks with fine-grained synchronization (GOODWIN, TID and E40R0100). Secondly, the improvement increases when the multiplexing degree increases. They show that our scheduler-conscious event waiting strategy scales well with the multiplexing degree and it is also more effective (up to 14% improvement) for applications with frequent synchronization.

MPI nodes	Multiplexing degree	Synchronization	Blocking	$\frac{Blocking}{Synchronization}$
4 nodes	1	23273 times	709 times (83 times/sec)	3.0%
8 nodes	2	29097 times	4227 times (316 times/sec)	14.5%
12 nodes	3	35868 times	9202 times (407 times/sec)	25.7%

Table 7: The total number of synchronization and blocking for running GOODWIN with TMPI-2 and $2ms$ spin time on a 4-processor Power Challenge.

The above improvement ratio is not so large and we expect that the improvement ratio may become larger for a large-scale SMM with a deeper memory hierarchy since data locality becomes more important; however we have not been able to find a larger number of dedicated Origin 2000 nodes to verify this statement and we intend to do so in the future.

# of MPI node	4	6	8	10	12
Multiplexing degree	1	1.5	2	2.5	3
GE	0.3%	2.4%	2.4%	3.0%	4.6%
SWEEP3D	0%	0.9%	2.1%	1.7%	6.1%
GOODWIN	0.6%	0.1%	8.3%	11.7%	14.4%
TID	0%	2.7%	5.2%	6.7%	12.6%
E40R0100	1.5%	1.1%	6.1%	8.1%	11.4%

Table 8: Performance improvement of scheduler-conscious event waiting over simple spin-then-block on an SGI Power Challenge.

6 Conclusions and related work

The main contribution of our work is the development of adaptive two-level thread management for optimizing the execution of MPI code on shared memory machines. It also includes a scheduler-conscious event waiting strategy to minimize spinning overhead and exploit cache affinity. The major advantage of this two-level management includes: 1) saving context switch cost and reducing cost for the spin-block type of synchronization during MPI communication; 2) exposing the thread scheduling information at user-level, which enables better collaboration between scheduling and synchronization. The experiments indicate that the proposed techniques can obtain great performance gains over our previous design and over SGI MPI in multiprogrammed environments (161% or more substantial improvements depending on the degree of multiprogramming) and is competitive in dedicated environments. The comparison of TMPI with MPICH is reported in [27] and TMPI can outperform MPICH significantly in a multiprogrammed environment. Thus we do not report the comparison of TMPI-2 with MPICH in this paper.

Our study is leveraged by previous research in OS job scheduling on multiprogrammed SMMs [8, 18, 29,

# of MPI node	4	6	8	10	12
Multiplexing degree	1	1.5	2	2.5	3
GE	0%	3.1%	2.2%	1.0%	3.4%
SWEEP3D	-1.0%	3.8%	1.1%	0.7%	3.7%
GOODWIN	0%	0.1%	7.5%	9.3%	14.2%
TID	0.2%	0.7%	5.3%	4.5%	8.2%
E40R0100	0.7%	0.4%	2.2%	4.8%	12.0%

Table 9: Performance improvement of scheduler-conscious event waiting over simple spin-then-block on an SGI Origin 2000.

32, 33]. These studies show that multiprogramming makes efficient use of system resources and space/time sharing is the most viable solution, outperforming other alternatives such as gang/co-scheduling for achieving high throughput and fast response times. The current version of OS in both SGI and SUN multiprocessors support space/time sharing policies. It is argued in [10] that gang-scheduling may be more practical since dynamic space slicing is not easy. SGI OS adopts gang-scheduling in IRIX 6.4; however IRIX 6.5 changed the default scheduling of shared memory applications, which allows dynamic space/time sharing. SGI made this change because a gang-scheduled job cannot run until sufficient processors are available so that all members of the gang can be scheduled, and the turnaround time for a gang-scheduled job can be slow [3]. Also in IRIX 6.5, gang-scheduled jobs do not get priority over non-gang scheduled jobs. SGI MPI in IRIX 6.5 uses the default OS scheduling (which is not gang-scheduling) and does not allow user-specified gang-scheduling (a mechanism that turns on gang-scheduling using `schedctl()` for an SPROC job does not work for this new SGI MPI version) [24]. Our goal is to allow an MPI parallel program to perform well in the presence of multiprogramming under different space/time scheduling policies.

The concept of a two-level thread structure can be found in SUN Solaris [22] where kernel threads are light weight processes. Our focus is on adaptive management of kernel threads so that the number of kernel context switches is minimized. Our adaptive thread control is motivated by the previous work on process control and scheduler activation [5, 29, 31] and the difference is that we focus on the tradeoff and adaptive control of kernel/user threads in executing MPI programs. The scheduler activation [5] and new OS research prototypes (e.g. Exokernel [9]) allow application-specific resource management and our techniques can be integrated with these systems if they are available commercially. In terms of adaptability to the current commercial systems, these techniques require substantial OS changes while our approach requires a very small change in the current commercial OS interface to expose the kernel scheduling decision or no changes if we can use a user-level central monitor to control all parallel jobs. Our scheduler-conscious work is motivated by an earlier study in [17] while their work was focused on locks and barriers and our strategy is built on a two-level thread scheme and considers cache affinity.

The work on multi-threading for parallelizing compilers is studied in [20, 21, 28, 31] and their threads are targeted at compiler generated fine-grained parallelism with simple synchronization while our threads are designed for running coarse-grained MPI programs with rich synchronization semantics. The thread system studied for the Cilk language [6] addresses lock-free management and adaptive mapping of user-level threads to OS kernel threads. Our study targets at MPI which has more complicated communication and synchronization primitives. The granularity of a thread in Cilk is typically much finer than that in

MPI, thus our work focuses less on work-stealing and more on context switch and cache affinity.

Acknowledgment

This work was supported in part by NSF CCR-9702640. We would like to thank Bill Gropp, Eric Salo, and anonymous referees for their helpful comments, and Claus Jeppesen for his help in using Origin 2000 at UCSB.

References

[1] Information Power Grid. http://ipg.arc.nasa.gov/.

[2] MPI Forum. http://www.mpi-forum.org/.

[3] NCSA note on SGI Origin 2000 IRIX 6.5. http://www.ncsa.uiuc.edu/SCD/Consulting/Tips/Scheduler.html.

[4] T.E. Anderson. The Performance of Spin Lock Alternatives for Shared-memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[5] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levey. Scheduler Activations: Effective Kernel Support for User-level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[6] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, June 1998.

[7] H. Casanova and J. Dongarra. Netsolve: a network server for solving computational science problems. *Proceedings of Supercomputing'96*, November 1996.

[8] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on Multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, December 1991.

[9] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of ACM SOSP'95*, 1995.

[10] D. Feitelson. Job Scheduling in Multiprogrammed Parallel Systems. Technical Report Research Report RC 19790, IBM, 1997.

[11] I. Foster and C. Kesselman (Eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[12] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[13] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[14] B. Jiang, S. Richman, K. Shen, and T. Yang. Efficient Sparse LU Factorization with Lazy Space Allocation. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[15] A. Karlin, K. Li, M. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proc. of ACM Symp. on Operating Systems Principles*, pages 41–55, October 1991.

[16] A. R. Karlin, M. S. Manasse, L. McGeoch, and S. Owicki. Competitive Randomized Algorithms for Non-Uniform Problems. In *Proceedings of the first ACM Symposium on Discrete Algorithms*, pages 301–309, October 1989.

[17] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-Conscious Synchronization. *ACM Transactions on Computer Systems*, 1997.

[18] S. T. Leutenegger and M. K. Vernon. Performance of Multiprogrammed Multiprocessor Scheduling Algorithms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1990.

[19] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the Distributed Computing Systems Conf.*, pages 22–30, 1982.

[20] C. Polychronopoulos, N. Bitar, and S. Kleiman. NanoThreads: A User-Level Threads Architecture. Technical Report CSRD #1297, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1993.

[21] E. D. Polychronopoulos, X. Martorell, and D. S. Nikolopoulos. Kernel-Level Scheduling for the Nano-Threads Programming Model. In *Proceedings of 1998 International Conference on Supercomputing*, pages 337–344, July 1998.

[22] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS Multi-thread Architecture. In *Proceedings of the Winter 1991 USENIX Conference*, 1991.

[23] B. Protopopov and A. Skjellum. A Multi-threaded Message Passing Interface(MPI) Architecture: Performance and Program Issues. Technical report, Computer Science Department, Mississippi State University, 1998.

[24] E. Salo. Personal Communication, 1998, 1999.

[25] K. Shen, X. Jiao, and T. Yang. Elimination Forest Guided 2D Sparse LU Factorization. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 5–15, June 1998. Available on the World Wide Web at $http://www.cs.ucsb.edu/research/S+/$.

[26] M. Snir, S. Ottto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[27] H. Tang, K. Shen, and T. Yang. Compile/Run-time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 107–118, May 1999.

[28] X. Tang and G. R. Gao. How "hard" is Thread Partitioning and How "bad" is a List Scheduling Based Partitioning Algorithm? In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 130–139, June 1998.

[29] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-memory Multiprocessors. In *the 12th ACM Symposium on Operating System Principles*, December 1989.

[30] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *Proceedings of ACM Symposium on Operating System Principles*, October 1991.

[31] K. K. Yue and D. J. Lilja. An Effective Processor Allocation Strategy for Multiprogrammed Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(12), December 1997.

[32] K. K. Yue and D. J. Lilja. Dynamic Processor Allocation with the Solaris Operating System. In *Proceedings of the International Parallel Processing Symposium*, April 1998.

[33] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 214–225, May 1990.