UCRL-JC-133263
PREPRINT

Benchmarking Pthreads Performance

B.R. de Supinski
J. May

This paper was prepared for submittal to the
1999 International Conference on
Parallel and Distributed Processing Techniques and Applications
Las Vegas, NV
June 28-July 1, 1999

April 27, 1999

Benchmarking Pthreads Performance

Bronis R. de Supinski
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
bronis@llnl.gov

John May
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
johnmay@llnl.gov

Abstract: The importance of the performance of threads libraries is growing as clusters of shared memory machines become more popular. POSIX threads, or Pthreads, is an industry threads library standard. We have implemented the first Pthreads benchmark suite. In addition to measuring basic thread functions, such as thread creation, we apply the LogP model to standard Pthreads communication mechanisms. We present the results of our tests for several hardware platforms. These results demonstrate that the performance of existing Pthreads implementations varies widely; parts of nearly all of these implementations could be further optimized.

1. Introduction

With the growing popularity of symmetric multiprocessors (SMPs), the importance of the performance of Pthreads libraries is increasing. However, no Pthreads benchmark suite currently exists. We are developing a benchmark suite that will fill this void. This tool will be useful in predicting and identifying performance problems of codes that use Pthreads.

We are modifying a publicly available MPI benchmark suite in order to measure Pthreads performance. This approach will allow us to eventually provide a benchmark suite for measuring the performance of mixed programming models for clusters of SMPs that use both threads and message passing. Our initial results from several SMP systems demonstrate significant performance differences between existing Pthreads implementations. Since hardware differences do not fully explain these performance variations, optimizations could improve the implementations.

2. Incorporating Threads Benchmarks into SKaMPI

SKaMPI is an MPI benchmark suite that provides a general framework for performance analysis [7]. SKaMPI does not exhaustively test the MPI standard. Instead, it provides a simple interface to incorporate additional measurements. This interface provides extensive facilities for data collection and test management, such as dynamic selection of independent variable values and of the number of trials to obtain an accurate measurement at any single data point. Thus, SKaMPI is an excellent starting point for implementing our Pthreads benchmark suite.

Nonetheless, several aspects of SKaMPI are inappropriate for a Pthreads benchmark suite. Since clock granularity varies widely across systems, most benchmarks time repeated measurement actions, such as locking a mutex, and use the average time per iteration to estimate the time that the action takes. SKaMPI does not; instead it assumes that the duration of an action is sufficiently long to be measured individually. Since this assumption is clearly inappropriate for several important Pthreads actions, we

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract no. W-7405-Eng-48.

have modified SKaMPI's data collection facilities to support multiple iterations of the measurement action per timing.

The number of timings per data point varies in SKaMPI. A measurement is repeated until either a user-defined maximum number of timings or the standard deviation[1] of the timings is less than a user-defined percentage of their mean. The mean is the reported measurement. This simple mechanism ensures data points are a good estimate of the average time to complete the measured action. Several of our benchmarks are symmetric. In these tests, the main thread and a client thread both repeatedly perform (essentially) the same action. All of our benchmarks measure the elapsed time in the main SKaMPI thread. In order to accommodate the dynamic number of timings per data point, we terminate the client thread with `pthread_cancel` in these tests instead of exiting after a fixed number of iterations. This cancellation mechanism introduces very little overhead into the tests. If the client action does not include a cancellation point (as defined by Pthreads), the client calls `pthread_test_cancel` after some large number of repetitions. The main thread does not start the next measurement until the client acknowledges the cancel.

A significant factor in the performance of many Pthreads functions is whether or not the threads being measured are running on the same CPU. Most vendors include a mechanism to bind a thread to a specific CPU. We implemented a set of macros that allow us to vary the CPU binding of each participating thread in each benchmark.

1. SKaMPI actually uses standard error, which is the standard deviation divided by the square root of the number of timings, for the cutoff. Standard error converges even when the standard deviation large. We use standard deviation since it provides a stricter test and the number of timings is already limited.

Clock granularity motivates another change to SKaMPI. Since SKaMPI is an MPI benchmark suite, it uses the `MPI_Wtime` facility for its timings. However, the MPI standard does not require that `MPI_Wtime` use the best available clock; on some systems, its resolution is as high as one millisecond. Since system resources can severely restrict the number of iterations per timing for some Pthreads calls, we are replacing `MPI_Wtime` with macro-based calls to more fine-grained clocks, which on many systems are the UNIX `gettimeofday` facility. However, on DEC Alpha systems, we are implementing a rollover-safe clock based on the hardware cycle counter.

We have made several other changes to the SKaMPI infrastructure, including several bug fixes to its data point selection and test restart facilities. We have also extended it to support more than one independent variable per data point, a facility that we use to ensure threads are running on different CPUs. We plan eventually to provide a compiler switch that eliminates the need to link with an MPI library; currently we simply run our threads tests within a single MPI process. However, we plan to retain the MPI tests, as our overall goal is a benchmark suite for an emerging programming model for SMP clusters in which threads on the same node communicate through shared memory and use message passing for internode communication [5].

3. The Benchmarks

Our benchmarks measure several significant components of Pthreads performance. Some measure basic system properties such as the time for thread creation or the length of a timeslice. The remainder of our tests measure thread communication and synchronization mechanisms, such as condition variables and mutexes. We describe each benchmark in detail.

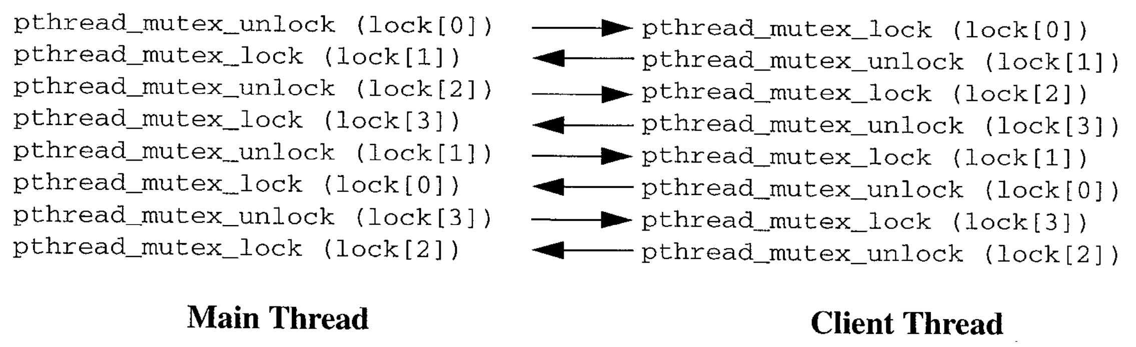
Our `pthread_create` benchmark measures the time to recursively create M threads, where M is some large number. In this benchmark, the main thread sets a shared iteration count to zero, creates a thread and waits on a condition variable. Each newly created thread increments the iteration count and compares it to M. If the count is M, the thread signals the main thread. Otherwise, the thread creates another thread and immediately calls `pthread_exit`. The measurement reported for this benchmark is the time taken by the main thread divided by M. We report results for when the threads are created either detached or undetached with either system or process scope.

Our thread yield test attempts to measure the time required for a context switch, similarly to one of the tests described by Mueller [6]. In this test, two threads repeatedly call `sched_yield` (`thr_yield` on Sun platforms). When the threads are bound to the same CPU, we expect the threads to alternate using the CPU and, thus, the total time divided by the number of yields approximates the thread context switch time. However, an auxiliary program revealed that this is not always the case under AIX. Under AIX, the initial thread does not regain the processor until its child thread has completed its yields. Therefore, on the IBM platform, this test first creates a proxy thread that measures the context switch time. With this mechanism, the threads alternate using the CPU as desired.

We also bind two threads to the same CPU in order to determine the length of a time slice. In this test, the main thread sets a shared variable to zero and spins until the variable is not zero. Similarly, the client thread sets the variable to one and spins until it is not one. Both threads repeat this behavior M times. Each thread begins its time slice by setting the shared variable and then expends the rest of its time slice spinning. Thus, the length of a time slice is the total time for this test divided by $2M$.

The Pthreads API provides several mechanisms for communication between threads. Our original intent was to apply the LogP model to each of these mechanisms [2]. LogP models message passing communication costs with four parameters: latency (L), which equals the time a message actually spends in transit; the overhead of sending (o_S) or receiving (o_R) a message; the gap, g, which is the minimum interval between message sends (or receives); and the number of processors, P. We present benchmarks that measure the round trip time ($2*(o_S + L + o_R)$) and the overheads for Pthreads communication through both mutexes and condition variables. Our initial tests, based on the method for measuring message passing send overheads [3], indicates that the gap does not exceed the "send" overhead for Pthreads communication mechanisms. We leave lower level tests that might reveal excess cycles for future work.

Condition variables provide an efficient mechanism for thread synchronization. A thread that is waiting on a condition variable can be suspended until it receives a signal of the condition. A thread must acquire a mutex that protects the condition before waiting on it. The mutex is released within the call to `pthread_condition_wait`; it is reacquired before the thread returns from the call, but after the signal is received. These semantics make a condition variable ping-pong test straightforward: each thread alternates between waiting on the condition and signaling it. Since a thread can send a signal even when no other thread is waiting for it (in which case the signal is discarded), the time to call `pthread_cond_signal` repeatedly in one thread measures the cost of signaling a condition, which is the "send" overhead of this communication. Potentially we could measure the receive overhead by waiting on a condition that another thread is



```
pthread_mutex_unlock (lock[0])  ------>  pthread_mutex_lock (lock[0])
pthread_mutex_lock (lock[1])    <------  pthread_mutex_unlock (lock[1])
pthread_mutex_unlock (lock[2])  ------>  pthread_mutex_lock (lock[2])
pthread_mutex_lock (lock[3])    <------  pthread_mutex_unlock (lock[3])
pthread_mutex_unlock (lock[1])  ------>  pthread_mutex_lock (lock[1])
pthread_mutex_lock (lock[0])    <------  pthread_mutex_unlock (lock[0])
pthread_mutex_unlock (lock[3])  ------>  pthread_mutex_lock (lock[3])
pthread_mutex_lock (lock[2])    <------  pthread_mutex_unlock (lock[2])
```

Figure 1: Mutex Ping-Pong Actions (arrows show operation order)

continually signaling since the signaling thread does not need to own the mutex. This mechanism requires that the waiting and signaling threads be bound to different CPUs; otherwise the test tends to measure the length of a time slice. Unfortunately, cache coherence and memory bus contention overheads can increase the time required for receiving the signal, as the results that we present for the IBM platform indicate. We leave investigating these overheads for future work.

We have also implemented a ping-pong test for mutex variables. In this test, we create four mutex variables, using the default protocol. Initially, the main thread holds the even indexed locks; the client the odd indexed ones. We ensure the main thread begins each ping-pong and that each pair of actions leaves the locks in their initial states by ordering the unlock/lock operations as shown by the arrows in Figure 1. Thus, the time for a pair of actions divided by four is the time required to perform one mutex ping-pong.

We have implemented several tests that measure the overheads of the mutex operations. First, we ran our mutex ping-pong test with the lock/unlock operations performed only in the main thread. This no contention test provides a reasonable estimate of the sum of the overheads for the mutex operations. Similarly to Mueller [6], our mutex lock and unlock test provides an alternative estimate of this sum by successively locking and unlocking the same mutex in the main thread; our results indicate that the no contention approach can underestimate the overheads due to ILP effects. Finally, we measure the individual overheads by first measuring the time to lock a large array of mutexes and then the time to unlock the same mutex array. Note that the array indexing overheads counteract any the ILP benefits. The overall effect varies across systems.

4. Comparison of SMP Systems

Table 1 shows results for our benchmark suite on three different platforms. Our Digital results are from a cluster of AlphaServer 4100's running OSF1 V4.0. Each node of this machine has four 533Mhz Alpha ev5 CPUs. Our IBM results are from LLNL's Combined Technology Refresh (CTR) SP2, running AIX V4.3.2. Each node of this machine has four 332 Mhz PowerPC 604e CPUs. Our SGI results are from Los Alamos National Laboratory's Nirvana Blue. Each node of this machine has up to 128 250 Mhz R10000 processors. Finally, we have obtained initial numbers for Sun's implementation of Pthreads. The Sun numbers are from an Enterprise 4000 with eight 168 Mhz UltraSparc Is CPUs. This general purpose interactive server at LLNL supports many users and can be quite busy. However, it was relatively quiet during our measurements. Nonetheless we are unable to perform the

Table 1: Benchmark Results (in μs)

Test		Digital	IBM	SGI	Sun
Thread Create	Detached, system scope	N/A	265.9	N/A	211.0
	Detached, process scope	N/A	265.5	30.6	51.7
	Undetached, system scope	N/A	312.7	N/A	491.5
	Undetached, process scope	N/A	305.1	68.9	343.1
Thread Yield		1.6	4.3	N/A	9.5
Time slice		10013	10001	N/A	N/A
Condition Ping-pong	Unbound	59.5	48.9	9.5	25.0
	Same CPU	12.4	29.2	N/A	25.7
	Different CPUs	N/A	74.3	N/A	25.1
Condition Signal		0.0383	0.606	0.634	0.125
Condition Wait		25.3	45.7	N/A	N/A
Mutex Ping-pong	Unbound	58.8	3.7	9.4	24.3
	Same CPU	13.4	37.8	N/A	24.2
	Different CPUs	N/A	3.7	N/A	24.2
	No Contention	0.274	0.638	0.434	0.448
Mutex lock and unlock		0.273	0.726	0.540	0.714
Mutex lock		0.483	0.343	0.251	0.321
Mutex unlock		0.495	0.478	0.296	0.417

time slice test reliably since it is a shared resource. The nodes on the DEC cluster are also shared, although we were able to run most of our tests when no users were active. Our IBM numbers are from a dedicated node, while the SGI machine has facilities to provide dedicated CPUs on a node.

Differences in the binding facilities of the machines significantly affect the tests that we are able to run. OSF1 only provides a facility to bind all of a process's threads to the same CPU; thus we do not report Digital results for threads bound to different CPUs. The situation on the SGI machine is worse. The man pages indicate that the binding facility works at the process level. However, testing indicates that it does apply to individual threads. Unfortunately, testing also reveals that the operating system can override the CPU binding after a small period of time, particularly when the threads are bound to the same CPU. Thus, we report SGI results only for the tests that use unbound threads.

Our numbers indicate significant differences between the Pthreads implementations. The SGI machine has excellent performance for creating threads with process scope - although its has the second slowest processors. The Sun machine also creates process scope threads quickly - if they are detached. Creating a thread as undetached allows the creating process to receive a return status when the created thread exits. Since this capability requires extra thread table state, undetached threads are more expensive to create, particularly on the Sun platform. Our tests with system scope threads hung on the SGI machine.

Although the Alphas are easily the fastest CPUs, the Digital machine is the slowest for our mutex and condition ping-pong tests with unbound threads. Since we expect unbound threads usually run on different CPUs, this result indicates the Digital Pthreads implementation needs to be better optimized for threads running on different CPUs. Also, Sun significantly outperforms IBM for the condition ping-pong test with threads bound to different CPUs although the IBM runs at essentially twice the clock speed. This result does not seem to reflect an inherent difference between the memory/bus systems of the machines since IBM's performance on the mutex ping-pong test with threads bound to different CPUs is excellent and significantly better than either of the other machines. We plan to use either lmbench or hbench:OS to explore differences between the memory systems further [4, 1]. However, we believe the differences in our results primarily arise in the Pthreads implementations.

5. Conclusions and Future Work

No Pthreads benchmark suite currently exists; we have begun implementing a set of benchmarks to fill this void. After further testing and refinement, we will make this Pthreads benchmark suite publicly available. Our initial results show that system hardware differences do not completely explain significant differences between existing Pthreads implementations.

We plan to expand our tests of Pthreads functionality. Additional benchmarks will measure the performance of functions such as pthread_cond_broadcast and pthread_cancel. We also are considering tests that evaluate higher level functionality that can be easily synthesized from Pthreads primitives, such as semaphores or barriers. We are also interested in the effect of using different mutex protocols, although many implementations currently support only the default protocol.

This work has arisen during a project evaluating the use of mixed programming models on clusters of SMPs. Our overall goal is to provide a benchmark suite that can measure important aspects of mixed model performance, such as the cost of a barrier across all threads in all MPI processes. We anticipate eventually extending this benchmark suite to measure the performance of OpenMP and other SMP compiler directives. Similarly to lmbench's context switch test, these tests will require a work mechanism that we also expect to use with additional Pthreads tests.

6. References

[1] A.B. Brown and M.I. Seltzer, "Operating System Benchmarking in the wake of *Lmbench*: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," *Proc. of the 1997 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1997, pp. 214-224.

[2] D.E. Culler, L.T. Liu, R.P. Martin and C.O. Yoshikawa, "Assessing Fast Network Interfaces," D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *Proc. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1993, pp. 1-12.

[3] D.E. Culler, L.T. Liu, R.P. Martin and C.O. Yoshikawa, “Assessing Fast Network Interfaces,” *IEEE Micro*, 1996, Vol. 16, No. 1, pp. 35-43.

[4] L. McVoy and C. Staelin, “lmbench: Portable tools for performance analysis,” *Proc. of the 1996 USENIX Technical Conf.*, 1996, pp. 279-295.

[5] J. May and B.R. de Supinski, “Experience with Mixed MPI/Threaded Programming Models,” *High Performance Scientific Computation with Applications, Technical Session of the 1999 Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, 1999, submitted.

[6] F. Mueller, “A Library Implementation of POSIX Threads under UNIX,” *Proc. of the 1993 Winter USENIX Conf.*, 1993, pp. 29-42.

[7] R.H. Reussner, “User Manual of SKaMPI, Special Karlsruher MPI-Benchmark,” *Tech. Report*, University of Karlsruhe, 1998.