Message-Passing Concurrency on Shared-Memory Multiprocessors

Charles Eric LaForest
laforest@eecg.utoronto.ca

December 20, 2007

Abstract

This paper presents an exploration of message-passing concurrency on a commodity Linux shared-memory multiprocessor. While message-passing has more overhead than shared-memory multithreading, it exhibits a greater scalability. Much of this desirable property is derived from the generality and ease of programming when using message-passing.

The message-passing system shown completely avoids shared memory between processes and the problems associated with correctly locking access to such memory. It demonstrates how to support thousands of processes, reduces the amount of copying between processes when sending messages, and schedules the processes efficiently to avoid waiting on locks held by sleeping processes.

The given micro-benchmark has thousands of processes contending for a single lock on a simple FIFO queue. However, because of clever process scheduling, the naive queue outperforms more complex implementations, including non-blocking ones.

1 Introduction

Message-passing has been in use for quite some time to program distributed computing systems, MPI (Message Passing Interface) being a well-known standard aimed at scientific computing spread across multiple machines.

Another example of message-passing concurrency can be found in the Ericsson AXD 301 ATM switch, controlled by software written in the concurrent programming language Erlang. Despite containing almost half-a-million lines of concurrent code describing complex telecommunication protocols, this telephony system has virtually no downtime, can recover from crashed processes, and can hot-swap code without stopping operations.

This amazing robustness suggests that message-passing concurrency is worth further investigation as a solution to the difficulty of creating correct, reliable parallel programs. What may be lost in absolute performance may be regained in scalability and ease of programming.

2 Related Work

Message-passing originated in 1972 in the Smalltalk programming language as a way of generalizing Simula's object-oriented system [8], without concern for actual concurrency.

In 1985, researchers at Ericsson began work on a language for describing telephony using the same model as Smalltalk [1]. This work resulted in the Erlang programming language which is used to implement large telephony switching systems. Although implemented in a VM for portability, there has been recent work in compiling to native code [7] as well as improving the efficiency of inter-process communication [13].

Pure message-passing implies that the contents of a message must be copied between processes. However, given a shared-memory architecture and some simple precautions, most of the copying can be avoided as in the IO-Lite buffering system [12].

Scalability in a message-passing system depends on the possibility of running a number of processes far larger than usually found in conventional systems. Past research on the topic [14] shows that the key feature needed to support a large number of processes is to keep the size of their

stacks to a minimum, else the virtual memory addressing space can get rapidly exhausted.

Finally, with many processes, the time-sharing abstraction of the underlying operating system breaks down. The performance of a software system is intimately tied to the scheduling of its parts [5], including locking mechanisms used in concurrent programming.

3 Design

The basis for the given message-passing system is shown in Figure 1. It is a conventional shared-memory multiprocessor system with the restriction that ordinary user-level processes may not access any memory external to their allocated area. Without any shared memory between processes, there can be none of the problems usually addressed with locks or transactional memory for example.

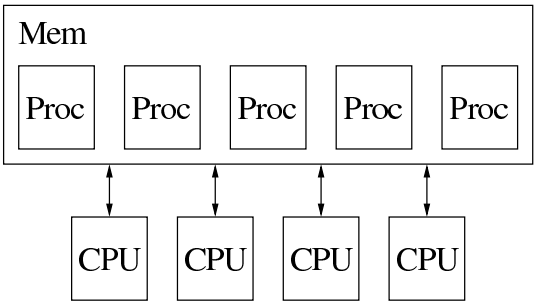

Figure 1: A shared-memory multiprocessor in which each process can only access its own memory.

The operating system kernel, however, can access the entire system memory. In order for a process to communicate with another, it must hand a message to the kernel via a system call which will then deliver the message to the specified recipient process. This is illustrated in Figure 2.

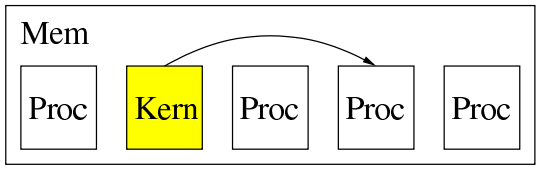

Figure 2: The kernel can see all memory and thus can deliver a message across processes.

To truly enable concurrency, multiple processes must be able to communicate simultaneously. This requires having multiple instances of the kernel active at once and reintroduces concurrent access to process memory. Errors are now possible when, for example, two separate processes attempt to send a message to a third process, or when a process must receive a message while sending or processing one, as seen in Figure 3. This reintroduces the need for mutual exclusion locks to access processes.

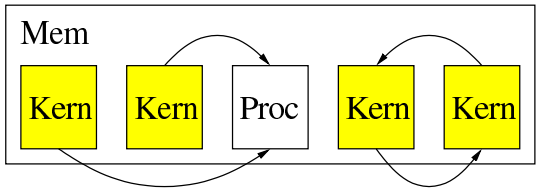

Figure 3: Multiple messages may be in flight and must be receivable no matter what the state of the recipient process.

3.1 Locks and Queues

Such locks are used as shown in Figure 4. Each process has an inbox message queue whose access is controlled by a mutex lock. There is also a similar free list queue to hold discarded messages in order to avoid the overhead of reallocating memory.

To receive a message, a process must briefly acquire the lock its inbox in order to pop the message at the head of the queue using the Recv() function. This gives the process a unique pointer to the message, eliminating any

possibility of contention while the process uses it. When the process is done, it Free()'s the message, placing it at the tail of the free list.

To send a message, a process Get()'s a message from the free list (or has one allocated if it is empty), prepares it, and passes it to the kernel via the Send() command. Upon successfully acquiring the lock to the inbox of the recipient process, the kernel can append the message to the tail of its queue.

Note that while sending a message requires a system call to the kernel, receiving one is only a user-space operation.

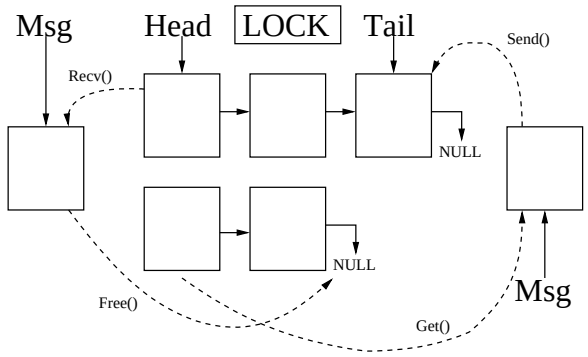

Figure 4: Structure of inbox and free message queues and the message-passing process.

3.2 Programmer's Perspective

While a mutex lock is required for the queue of each process, the programmer never sees it. This single fact accounts for much of the ease of programming in a message-passing concurrency model. To demonstrate this, let us take a simple example of one Receiver process (or several) performing computations on the behalf of a Sender.

The Sender (Algorithm 1) first obtains a message from its free list and prepares it with instructions to the Receiver (usually just a message type) and with information copied from local data. The Sender then sends the message to the Receiver via a system call. When the call returns, the Sender then waits for a message by trying to pop one from its inbox queue. Once the message is received, the Sender replaces the original local data with that returned by the Receiver.

Algorithm 1 Pseudo-code for Sender

```
Sender(){
 data_t local_data;
 msg_t *msg;
 msg = Get();
 Copy(&local_data, msg);
 Send(Receiver, msg);
 msg = Recv();
 Copy(msg, &local_data);
}
```

The Receiver (Algorithm 2) sits in an infinite loop which begins by waiting for a message. Once received, the Receiver processes the message based on its type, sends it back to the Sender, and resumes waiting for a message.

Algorithm 2 Pseudo code for Receiver

```
Receiver(){
 msg_t *msg;
 for(;;){
  msg = Recv();
  switch(msg->type){
   case type_one:
    Process(msg);
    break;
   case type_two:
    ...
  }
  Send(Sender, msg);
 }
}
```

This simple mechanism synchronizes the Sender and the Receiver without explicit locks.

4 Implementation

The design of this message passing system is implemented using the Pthread library built from the Linux Native POSIX Threads Library. The system maintains a global array containing all the data structures for each thread, representing the total system memory shown in Figure 1. Most of this data is solely for the Pthread library and is inaccessible to the threads themselves.

Processes Each process is implemented as thread which uses only local storage and whose main function arguments are a subset of its entry in the global thread array: the thread's identification number (which is its index into the global thread array), and a 'postoffice' consisting of the inbox and free list queues (Figure 4). Other than locally declared variables[1], this is the only data accessible by a thread.

The size of the thread stacks was kept to the minimum possible without experiencing segfaults. For the micro-benchmark used here, this size is 32kB, which is twice the minimum size allowed by the Pthread library.

Messages A message is composed of a body and a pointer. The pointer is used to link messages together in the postoffice queues. The body contains the type of the message and its contents. The type is used by the recipient to determine what to do with the message. To keep this system simple and avoid data packaging issues, the structure of the contents of a message is tailored to the application. In this case, the contents are the TID of the sending process and a structure containing an array, its length, and a flag specifying whether the array is already in sorted order.

Sending/Receiving The Send() system call is simulated by a plain function call whose parameters are the thread identification number (TID) of the recipient thread, and a pointer to the message to send. The function uses the TID to look up the address of the recipient's postoffice in the global thread array and appends the message to the inbox queue. The message pointer is not returned by the call and must be considered destroyed by the thread.

There are two versions of the Send() function. The first version obtains a free message from the recipient's postoffice, copies the message into it, and places it in the recipient's inbox. The original message is then freed to the sender's free list queue. In a system where processes absolutely cannot share memory by virtue of being in separate computers or within a non-uniform memory architecture, this copying is inevitable.

The second version takes advantage of the shared-memory nature of this system: Although processes cannot by convention access any memory beyond their own, there are no restrictions on where that memory may be located, so long as the process has a pointer to it[2]. The second version of Send() directly appends the sender's message to the recipient's inbox, thus only copying the value of a single pointer.

The Recv() function is not a system call since it only accesses the process' own postoffice. It sits in a loop which attempts to dequeue a message from the inbox and yields the process' current time slice if the queue is empty.

Postoffice Queues While the postoffice inbox and free list queues are implemented conventionally using singly-linked lists guarded by a mutex lock, the possibility of contention when reading or writing the queue suggests different approaches in order to try and avoid potential bottlenecks.

Pthread Mutex The initial implementation simply protects the entire queue with a single Pthread mutex lock. A queue is empty if both the head and tail pointers are NULL, which makes an empty queue a special case. Algorithms 3 and 4 implement this queue.

[1] Nothing prevents the use of globals or externs, but they are not required either, and better avoided.

[2] This was the driving reason to implement message-passing 'processes' as threads within a single operating system process.

Algorithm 3 Pthread Message Enqueue

```
void QueueMsg(mailbox_t *mailbox,
              msg_t *msg){
 pthread_mutex_lock(&mailbox->mutex);
 // If mailbox is empty
 if(mailbox->head == NULL){
  mailbox->head = msg;
  mailbox->tail = msg;
  msg->next = NULL;
 } else {
  msg->next = NULL;
  mailbox->tail->next = msg;
  mailbox->tail = msg;
 }
 pthread_mutex_unlock(&mailbox->mutex);
}
```

Algorithm 4 Pthread Message Dequeue

```
msg_t *DequeueMsg(mailbox_t *mailbox){
 msg_t *msg;
 pthread_mutex_lock(&mailbox->mutex);
 // If mailbox is empty
 if(mailbox->head == NULL){
  msg = NULL;
 } else {
  msg = mailbox->head;
  // If this is the last msg
  if(mailbox->head->next == NULL){
   mailbox->head = NULL;
   mailbox->tail = NULL;
  } else {
    mailbox->head = mailbox->head->next;
  }
 }
 pthread_mutex_unlock(&mailbox->mutex);
 return msg;
}
```

Non-Blocking Since multiple processes may be contending to access a queue, including the process which owns the queue, it is important to avoid the wasted time that arises when a scheduled process is waiting on a lock held by another currently sleeping process. Algorithms 5 and 6 implement a non-blocking queue which uses the Compare-And-Swap (CAS) atomic operation [11].

A particular aspect of this algorithm is that an empty queue holds a dummy message so that `head->next` always exists. This has the side effect that the message that is actually dequeued is the *previous* first message in the queue. The latest message is indicated by the `head` pointer. This behaviour forces a copy of the head message into a newly allocated message in order to preserve the original queue semantics and ensure that a message is referenced by only one pointer at any given time.

Normally, a version number must be kept along with a pointer manipulated by a CAS operation in order to avoid the ABA problem. This requires either a double-word CAS or reducing the pointer to an array index so the version number may be stored in the same word. This queue does not suffer from the ABA problem since there can never be multiple dequeues in progress: the owning process is the only reader and sending processes can only be writers.

From the writer's point of view, the `tail->next` pointer can never be changed and restored since it is either replaced by a new message or remains as the dummy message in an empty queue. As for the reader, the `head->next` pointer will never be changed and restored by another reader, so the case of an empty queue briefly containing a message cannot occur.

Algorithm 5 Non-Blocking Message Enqueue

```
void QueueMsg(mailbox_t *mailbox,
              msg_t *msg){
 msg_t *tail;
 msg_t *next;
 msg->next = NULL;
 for(;;){
  tail = mailbox->tail;
  next = tail->next;
  if(tail == mailbox->tail){
   if(next == NULL){
    if(CAS(&tail->next, next, msg)){
     break;
    }
   } else {
    CAS(&mailbox->tail, tail, next);
   }
  }
 }
 CAS(&mailbox->tail, tail, msg);
}
```

Algorithm 6 Non-Blocking Message Dequeue

```
msg_t *DequeueMsg(mailbox_t *mailbox){
 msg_t *msg;
 msg_t *head;
 msg_t *tail;
 msg_t *next;
 for(;;){
  head = mailbox->head;
  tail = mailbox->tail;
  next = head->next;
  if(head == mailbox->head){
   if(head == tail){
    if(next == NULL){
     return NULL;
    }
    CAS(&mailbox->tail, tail, next);
   } else {
    msg = CreateMsg();
    CopyMsg(next, msg);
    if(CAS(&mailbox->head, head, next)){
     break;
    }
   }
  }
 }
 DestroyMsg(head);
 return msg;
}
```

Test-and-Test-and-Set This queue is identical to the Pthread Mutex implementation, except that the mutex is implemented as a Test-and-Test-and-Set lock with exponential back-off. The lock itself is implemented in x86 assembly using Bit-Test and Bit-Test-and-Set instructions.

Pthread Mutex (Two Locks) A process receiving messages at a high rate could find itself unable to process them rapidly enough due to the fact that the inbox queue lock would almost always be held by a sending process. Making it possible to concurrently enqueue and dequeue messages would avoid this scenario.

Algorithms 7 and 8 implement a Two Lock queue [11] which enables the receiving process to extract messages from the head of the queue while sending processes are placing messages at its tail. It requires two ordinary mutex locks, implemented here as Pthread mutexes, one each for the `head` and `tail` pointers.

Like the Non-Blocking algorithm, this queue depends on there always being a message present in the queue, using a dummy when the queue is empty. This forces the dequeued message to be copied in order to keep the same external behaviour as the plain Pthread Mutex queue.

Algorithm 7 Pthread Two Lock Message Enqueue

```
void QueueMsg(mailbox_t *mailbox, msg_t *msg){
 msg->next = NULL;
 pthread_mutex_lock(&mailbox->tail_mutex);
 mailbox->tail->next = msg;
 mailbox->tail = msg;
 pthread_mutex_unlock(&mailbox->tail_mutex);
}
```

Algorithm 8 Pthread Two Lock Message Dequeue

```
msg_t *DequeueMsg(mailbox_t *mailbox){
 msg_t *msg;
 msg_t *node;
 msg_t *new_head;
 pthread_mutex_lock(&mailbox->head_mutex);
 node = mailbox->head;
 new_head = node->next;
 // If mailbox is empty
 if(new_head == NULL){
  pthread_mutex_unlock(&mailbox->head_mutex);
  msg = NULL;
  return msg;
 }
 mailbox->head = new_head;
 pthread_mutex_unlock(&mailbox->head_mutex);
 msg = CreateMsg();
 CopyMsg(new_head, msg);
 DestroyMsg(node);
 return msg;
}
```

Pthread Mutex (No Free List) In order to compare the impact of the lack of true dequeueing in the Two Locks and Non-Blocking queue implementations, a version of the postoffice was created without a free list queue. This forces the allocation, copying, and de-allocation of messages found in the aforementioned queues. This queue otherwise operates like a Pthread Mutex implementation.

5 Micro-Benchmark: Parallel Sort

The micro-benchmark used to demonstrate this message-passing system is a parallelization of the sorting of an array of integers. The sorting algorithm is Bubble Sort, chosen mainly for its simplicity and easy indication if an array is already sorted[3].

Figure 5 illustrates the overall structure of the micro-benchmark. A Controller process creates a Slicer process and a number of Sorter processes, using messages as handshakes for synchronization and verification that a process is alive. The Controller then sends a message to the Slicer, telling it to begin sorting the array. The Slicer will eventually reply to the Controller, signifying that the list is now sorted.

The Slicer proceeds by sending slices of the array to Sorter processes, which sort the slice and return it to the Slicer who then re-integrates it into the array. The slices are copies: no Sorter can see the array. It is evident that the bottleneck will be the Slicer, whose inbox queue will receive all the Sorter's replies.

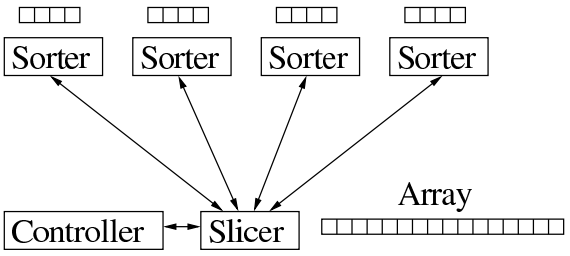

Figure 5: Structure of concurrent sorting micro-benchmark

The distribution of the array to the Sorter processes is done in the manner of the Even-Odd Sort[4]. Figures 6 and 7 illustrate the two phases of the process. The first phase evenly divides the array between all the Sorters, replacing the original slice with its sorted version.

[3]If a swap ever occurs, a flag is set.

[4]This also has the effect of reducing the sorting time from $O(n^2)$ to approximately $O(n)$. It's still not a very fast sort overall.

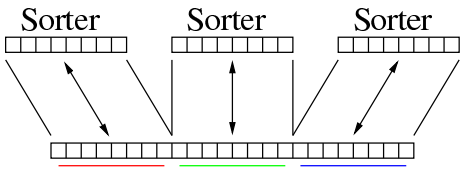

Figure 6: Even-Odd Bubble Sort Phase One

The second phase repeats this process, but shifts the location of the slices down by one-half their length. This has the effect of swapping the larger numbers from the lower slice with the smaller numbers from the next slice, given a left-to-right increasing sort order[5].

Given N Sorter processes, $2N$ phases are required to sort the entire array, sending a total of $(2N)^2$ messages of size $\frac{A}{N}$, given an array of size A.[6]

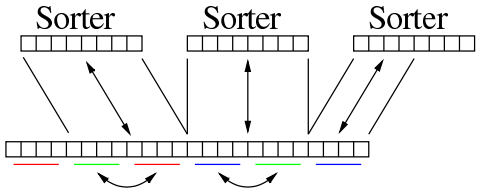

Figure 7: Even-Odd Bubble Sort Phase Two

The array is composed of 100,000 integers. The sorting process is repeated with varying numbers of Sorter processes. Before each run, the array is filled with the same pseudo-random number sequence whose numbers range from zero to the length of the array[7]. The length of a slice is the length of the array divided by the number of Sorter processes.

[5]The last Sorter could be omitted in the second phase since it moves nor sorts any data, but its overhead becomes insignificant given a large number of Sorters.

[6]The message content overhead (pointers, type, and sorted flags) is negligible except for extremely small messages.

[7]Generated by `rand() % length`, with a seed of `1`.

6 Results

The micro-benchmark was run over a number of Sorter processes, ranging from 1 to 10 in increments of one, 20 to 100 in increments of ten, 200 to 1000 in increments of 200, and 2000 to 6000 in increments of 2000.

Over this range, the micro-benchmark is run for the sequential case (one Sorter doing each slice in turn), the linking and copying versions of the Send() function, and both again using an 'empty' benchmark which does no sorting but transfers the same size and number of messages in order to highlight the overhead of message-passing.

The test platform was an 1.6GHz quad-Xeon (with SMT support) server running Linux 2.6.17.3 with GNU C Library version 2.3.6. The implementation of Pthreads was based on the Native POSIX Thread Library (NPTL) 2.3.6 [3].

6.1 Performance Overview

Overview Figures 8 through 12 show the overall performance for the five different message queue implementations. The most striking fact is that, over the entire range, the performance is essentially independent of the queue implementation! All versions show a maximum speedup of about three at about 100 Sorter processes, and exhibit runaway message-passing overhead at about the same point of 1000 Sorters (except for Test-and-Test-and-Set, which suffers a bit earlier).

Speedup A speedup of three can be considered differently depending on how one considers the architecture of the multiprocessor system. On the one hand, a speedup of three on a four-processor system is excellent. This is credible given that there are four physical processors sharing the memory bus and that the benchmark is memory-bound. On the other hand, the four Xeon processors use Simultaneous Multi-Threading (SMT) to appear as eight virtual processors, making a speedup of three seem less significant. However, the L1 cache and the load/store buffers are partitioned between the two virtual processors [6], halving the memory bandwidth available to each. In the end, the speedup is made possible by SMT: it allows for more processes to sort array slices (in cache) while others send/receive messages.

Copying Overhead It is also interesting to note that the Pthread Mutex implementation without a free list queue (Figure 10) shows the same lack of difference in performance between the link and copy versions as do the Non-Blocking and Two Locks versions, which force the allocation and copying of messages extracted from the queue. However, the difference between link and copy Send() is still apparent in Test-and-Test-and-Set despite its poor behaviour at high process counts, and also in Pthread Mutex. Overall, copying messages incurs only a small, constant overhead over linking.

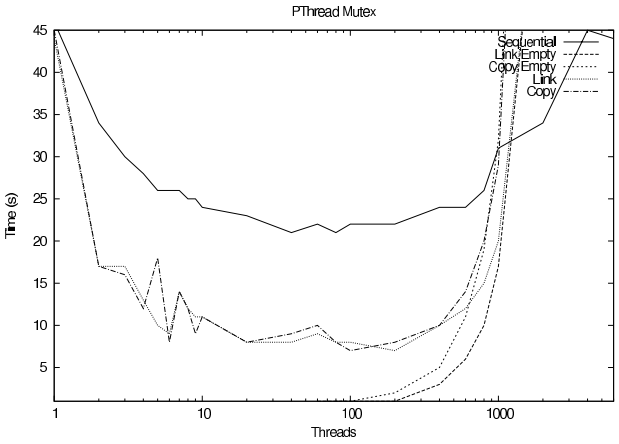

Figure 8: Performance overview using a queue managed by a single Pthread mutex

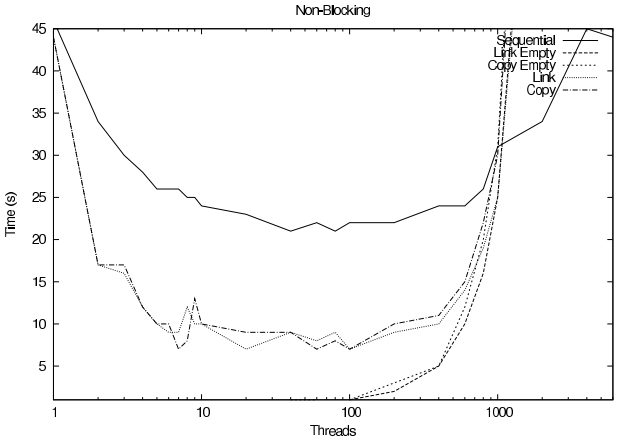

Figure 9: Performance overview using a non-blocking queue

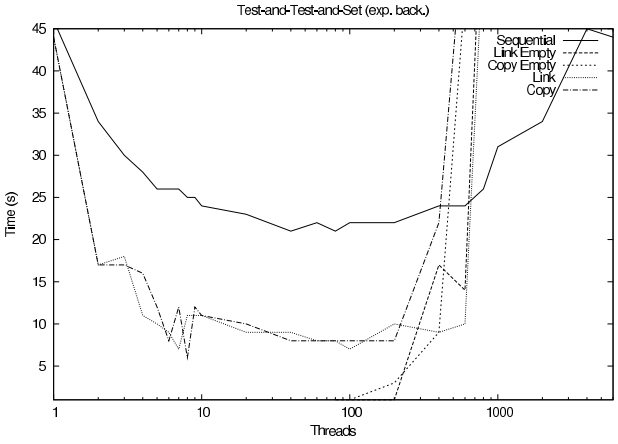

Figure 11: Performance overview using a test-and-test-and-set lock with exponential back-off

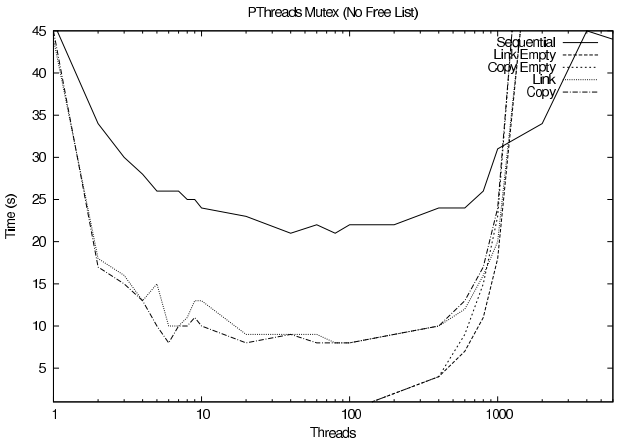

Figure 10: Performance overview using a queue managed by a single Pthread mutex, but without a free list queue, forcing malloc/free on messages.

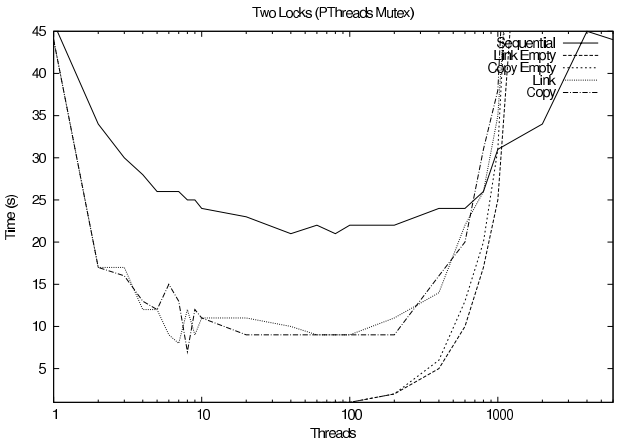

Figure 12: Performance overview using a queue managed with separate Head and Tail locks (Pthread mutexes)

6.2 Performance Under Heavy Contention

The differences in behaviour of the various implementations are not visible for a small number of threads (one to 100, approximately). To highlight the behaviour under significant stress, Figure 13 shows the linking Send() version of each of the queue implementations over the range of 100 to 1000 Sorter threads, while Figure 14 shows the same under pathological conditions for 1000 to 6000 threads[8].

Pthread Mutex Surprisingly, the simplest implementation exhibits the best performance, with a negligible loss if no free list is used, due to the `malloc()` function internally keeping a free list of previously allocated structures [9].

The underlying reason for the good performance is the use of Linux futexes ('fast userspace mutexes') to implement the Pthread mutexes [4][2]. Futexes behave like ordinary userspace mutexes when there is no contention, otherwise the contending process is appended by the kernel to a run queue attached to that futex. The net effect is that processes waiting for a lock obtain it in FIFO order without spinning on the actual lock. This effectively implements the MCS algorithm [10] as part of the OS scheduler, eliminating the problem of running processes waiting on locks held by sleeping processes.

Non-Blocking Contrary to expectation, the Non-Blocking queue is 25% slower than the simple Pthread Mutex implementation. This is primarily due to the heavier memory traffic it requires caused by multiple atomic CAS operations.

Test-and-Test-and-Set The Test-and-Test-and-Set queue fares well initially, but its breakdown is rapid beyond 600 Sorter threads. This is to be expected as it causes cache invalidation traffic every time a lock is released and heavy contention will cause the exponential back-off to reach excessive durations, both delaying processes and tying up processors in spin loops.

[8]For example, at 4000 processes, a message is only 100 bytes long, and 16 times as numerous as for 1000 processes.

Two Locks Finally, the Two Locks queue has the worst overall performance, contrary to past findings [11]. It is unclear why, as the algorithm is essentially the same as the Pthread Mutex implementation, except that a dummy message is kept on the queue. There was not enough time to analyze the compiled code further.

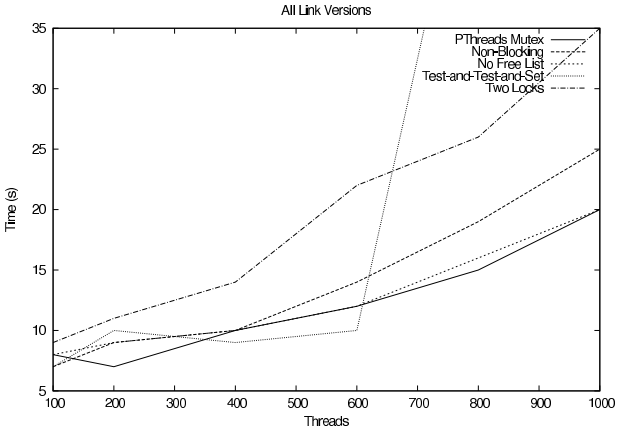

Figure 13: Comparison of several link benchmark versions from 100 to 1000 threads

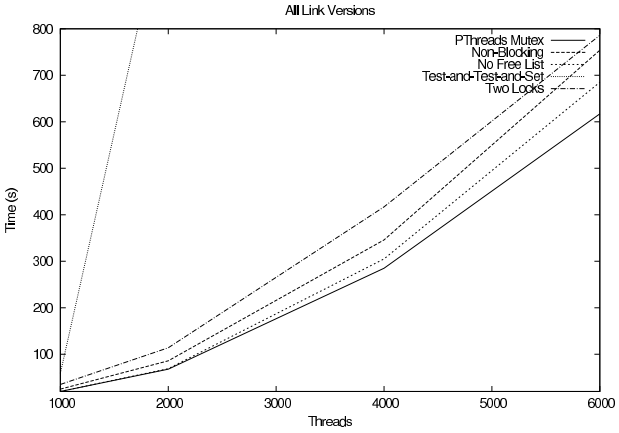

Figure 14: Comparison of several link benchmark versions from 1000 to 6000 threads

7 Conclusions

This report presents an exploration of a message-passing system implemented using Pthreads on a commodity Linux shared-memory multiprocessor. This system was used to implement a parallel sorting algorithm as a demonstration of the ease of programming using message-passing and as a micro-benchmark for the behaviour under heavy contention of a number of different message queue algorithms.

The benchmark results show that the performance of such a message-passing system is practically independent of the message queue implementation over a large range of number of contending processes, all versions achieving a speedup of about three over the sequential, single process case. Under extremely heavy contention, a queue protected by a single Pthread mutex gave the best performance, exceeding that of two lock and non-blocking versions, due to the underlying Linux futexes implementing an MCS-like scheduling algorithm.

Furthermore, the use of free lists has little impact on performance overall, and the copying of messages into a queue instead of their linking generates only a small, constant overhead.

These results suggest that despite the obvious performance penalty of message-passing without shared memory over conventional shared-memory multithreading, the ease of programming and the scalability of message-passing make it a viable option for reliably creating large concurrent systems.

8 Further Work

While this initial attempt at a message-passing framework was successful, there remain some unanswered questions and potential improvements:

Stacks While the stack memory allocated to a Pthread thread was reduced to 32kB down from the defaults of one to two megabytes, thus avoiding the exhaustion of virtual addressing space, this amount is still large compared to the few hundred bytes of stack for a small Erlang process. The Pthread stack is appended with a page without read/write permissions to catch overflows. The use of a segfault signal handler might allow for adjustable stacks as in Capriccio [14].

Thread Affinity No thread affinity was used in this system, leaving the decision to the OS. More careful selection of where threads run might improve performance through better cache locality.

Maximum Number Of Threads During the construction of this message-passing system, a limit of about 8000 threads was encountered beyond which thread creation slowed to a crawl. There was no apparent reason, but it might be an internal limitation of the Linux kernel.

Futex Signalling Although the current MCS-like behaviour of futexes provides excellent thread scheduling under lock contention, the ability to signal to the scheduler that the process queue attached to a given futex should be given preference next would further improve performance by waking processes the application knows have messages waiting for them.

References

[1] ARMSTRONG, J. A history of erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages* (New York, NY, USA, 2007), ACM, pp. 6–1–6–26.

[2] DREPPER, U. Futexes are tricky. Tech. rep., Red Hat, Inc., Raleigh, NC, USA, dec 2005. Online at: http://people.redhat.com/drepper/futex.pdf.

[3] DREPPER, U., AND MOLNAR, I. The native posix thread library for linux. Tech. rep., Red Hat, Inc., Raleigh, NC, USA, Feb. 2005.

[4] FRANKE, H., RUSSELL, R., AND KIRKWOOD, M. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the 2002 Ottawa Linux Summit* (June 2002), pp. 479–495.

[5] HARIZOPOULOS, S., AND AILAMAKI, A. Affinity scheduling in staged server architectures. Tech. Rep. TR CMU-CS-02-113, Carnegie Mellon University, Mar. 2002.

[6] INTEL CORPORATION. *Intel 64 and IA-32 Architectures Optimization Manual*. Denver, CO, USA, Nov. 2007.

[7] JOHANSSON, E., PETTERSSON, M., AND SAGONAS, K. A high performance erlang system. In *PPDP '00: Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming* (New York, NY, USA, 2000), ACM, pp. 32–43.

[8] KAY, A. C. The early history of smalltalk. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages* (New York, NY, USA, 1993), ACM, pp. 69–95.

[9] LEVER, C., AND BOREHAM, D. malloc() performance in a multithreaded linux environment. Tech. rep., Sun-Netscape Alliance, May 02 2000.

[10] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9*, 1 (1991), 21–65.

[11] MICHAEL, M. M., AND SCOTT, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 1996), ACM, pp. 267–275.

[12] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Io-lite: a unified i/o buffering and caching system. *ACM Trans. Comput. Syst. 18*, 1 (2000), 37–66.

[13] STENMAN, E., AND SAGONAS, K. On reducing interprocess communication overhead in concurrent programs. In *ERLANG '02: Proceedings of the 2002 ACM SIGPLAN workshop on Erlang* (New York, NY, USA, 2002), ACM, pp. 58–63.

[14] VON BEHREN, R., CONDIT, J., ZHOU, F., NECULA, G. C., AND BREWER, E. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), ACM, pp. 268–281.