Managing Concurrent Access for Shared Memory Active Messages *

Steven S. Lumetta and David E. Culler
Computer Science Division
University of California at Berkeley
{stevel,culler}@CS.Berkeley.EDU

Abstract

Passing messages through shared memory plays an important role on symmetric multiprocessors and on Clumps. The management of concurrent access to message queues is an important aspect of design for shared memory message-passing systems. Using both microbenchmarks and applications, this paper compares the performance of concurrent access algorithms for passing active messages on a Sun Enterprise 5000 server. The paper presents a new lock-free algorithm that provides many of the advantages of non-blocking algorithms while avoiding the overhead of true non-blocking behavior. The lock-free algorithm couples synchronization tightly to the data structure and demonstrates application performance superior to all others studied. The success of this algorithm implies that other practical problems might also benefit from a reexamination of the non-blocking literature.

1. Introduction

The ability to pass messages through shared memory plays an important role both in applications and in the operating system on symmetric multiprocessors (SMP's). The implicit synchronization encapsulated in the message abstraction simplifies the construction of otherwise asynchronous data transfers and provides an efficient mechanism for serializing operations on complex data structures. Messages are also a generally accepted tool for parallel programming and offer an attractive uniform interface for programming clusters of SMP's (Clumps).

Two aspects of design are particularly important to obtaining performance with shared memory message-passing protocols: the arrangement of data to reduce cache-coherence transactions and the management of concurrent access to message queues. Good solutions to the former require only the application of well-understood techniques from the literature, but efficient concurrent access can be challenging. Systems that approach this problem often sidestep concurrent access through the use of separate resources for each sender-receiver pair, but such solutions can hurt performance in a well-tuned, user-level communication layer.

In this paper, we study the performance of a variety of concurrent access algorithms for message-passing. Real applications are the primary tool for our investigation, but a set of microbenchmarks helps to characterize performance in the extremes of high and low contention. The platform for these tests is a shared memory version of Active Messages-II [16] that operates on a Sun Enterprise 5000 server running the Solaris 2.5 operating system. This system is part of a multi-protocol communication layer designed for Clumps [15].

The literature separates concurrent access algorithms into three disjoint groups. Traditional algorithms [1, 18, 20] are *locking*: a process must obtain a mutually exclusive lock to enter a critical section, thereby preventing other processes from entering concurrently. When such locks are used, a process stalled inside a critical section can delay all others for an arbitrary amount of time, a behavior termed blocking. *Non-blocking* algorithms [9, 10, 17, 19] hence guarantee that some process makes progress in a finite amount of time, which implies that they do not enforce mutual exclusion. The remaining algorithms do not use locks but can still result in blocking behavior [2, 12, 22]. We follow Valois [22] and adopt the term *lock-free* for this third category.

Non-blocking algorithms are advantageous on multiprogrammed systems, since locks interact poorly with time-sharing. These algorithms follow a common design strategy and are simpler than their optimized locking counterparts. A typical non-blocking operation works as follows. A process reads a value from a data structure, performs all computation based on the value read, and inserts the results atomically into the data structure. If another process has changed the original value during the computation phase, the computed

*The Enterprise 5000 server used in this work was donated by Sun Microsystems, Inc. This work was also supported in part by funding from National Science Foundation Infrastructure Grant CDA 94-01156, Lawrence Livermore National Laboratory Intra-University Transaction Agreement B336568, and the Defense Advanced Research Projects Administration Grant F30602-95-C-0014. The authors would also like to thank Andrea Dusseau, Arvind Krishnamurthy, Girija Narlikar, Kathy Yelick, and the anonymous reviewers for their insightful comments.

results are discarded and the operation starts again from the beginning. For many operations, the first steps of such an approach are the same as a sequential implementation, making non-blocking algorithms straightforward to construct. However, since most architectures support only 64-bit synchronization primitives, atomic insertion of the results often requires an extra level of indirection in the data structures.

On a dedicated system, the overhead of additional indirection and the cost of discarding optimistically completed work make generic non-blocking algorithms slower than locking algorithms. To address this drawback, numerous efforts apply problem-specific information to build more efficient solutions out of universal primitives. This optimization process sometimes involves sacrificing true non-blocking behavior in favor of fast common-case performance. The result is a lock-free algorithm. In practice, we expect these algorithms to provide many of the advantages of non-blocking algorithms while avoiding most non-blocking overheads. This paper presents a new lock-free algorithm for concurrent message queues and demonstrates performance superior to several locking alternatives.

The remainder of the paper is organized as follows: in the next section, we describe the problem domain and the experimental platform used in this study; Section 3 describes six concurrent access algorithms, including our lock-free approach; in Section 4, we evaluate performance using both microbenchmarks and applications; Section 5 provides information about related work; and Section 6 presents our conclusions.

2. Background and Environment

Two aspects of hardware performance are critical to shared memory message-passing: the memory hierarchy and the cost of synchronization primitives. The memory hierarchy governs the rate at which data moves from one processor to another, and synchronization primitives impose the basic overhead for concurrent access to data by multiple processes. Our experimental platform is a Sun Enterprise 5000 server containing eight 167 MHz UltraSPARC processors with 512 kB of L2 cache each and a total of 512 MB of main memory in two banks. Processors are connected via Sun's Gigaplane Interconnect [21], which is representative of many modern cache-coherent system interconnects. Its most unusual characteristic is support for more than one outstanding transaction on a single cache line, effectively pipelining concurrent memory traffic between processors. Cache coherence is maintained with an invalidation protocol. Memory accesses on the Enterprise incur 300 nanoseconds of latency, while obtaining a line from another processor's L2 cache requires 480 nanoseconds. The Enterprise supports both the universal [11] COMPARE&SWAP primitive (CAS in the text) and the less powerful TEST&SET primitive (T&S), each at a cost of roughly 90 nanoseconds for cached data.

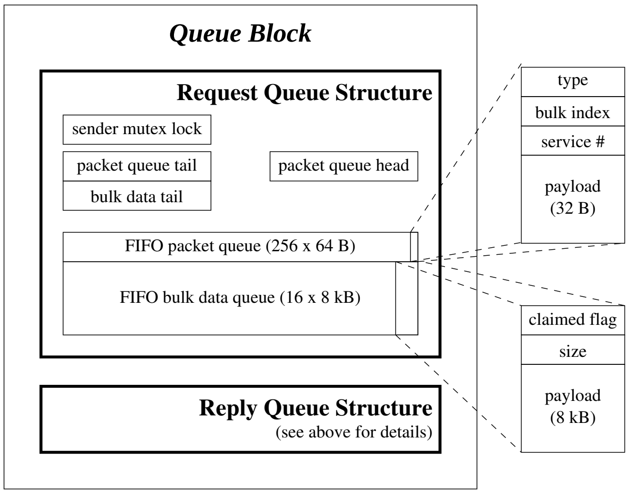

Figure 1. Block diagram of a queue block.

This work arose in the context of passing active messages through shared memory [15]. Active messages are similar to a highly-optimized RPC mechanism in which each communication *endpoint* acts as both client and server. Individual messages in the request-reply protocol can be either short messages of up to eight words or bulk data transfers of up to 8 kB. In this study, sets of processes communicate through single endpoints. Each endpoint statically allocates a System V shared memory segment to hold a *queue block* for incoming messages. Multiple clients operate concurrently on this block, requiring atomic enqueue operations for correctness. Processes poll their respective endpoints to detect the arrival of messages.

Our focus on message-passing between processes distinguishes this study from most literature on concurrent algorithms. Traditionally, these algorithms are evaluated with repeated calls to lock and unlock or to enqueue and dequeue. Although these tests do evaluate an algorithm's performance under high contention, they provides little intuition about application-level performance. In our message-passing system, for example, each concurrent operation is accompanied by uncontended computation—passing the message and handling it. A second difference in this work is the many-to-one nature of our queues: many processes send to a queue, but only one process receives from a queue. Finally, our use of a small, statically allocated block of memory restricts our choice of algorithms. Non-blocking algorithms cannot block when a queue is full and hence rely on dynamic allocation by their very nature.

The queue block for an endpoint appears in Figure 1. Two destination queue structures hold request and reply messages received by the endpoint. Each queue structure divides into three sections: queue tail information, accessed only by senders; queue head information, accessed only by the recipient; and two circular data queues, accessed by both senders and the recipient. Short messages use only the *packet queue*,

```
CLAIMPACKET(^q)
  while TRUE
    LOCK(q^.mutex)
    index ← q^.tail
    if q^.packet[index].type = FREE
      q^.packet[index].type ← CLAIMED
      q^.tail ← (index + 1) mod Q_LENGTH
      UNLOCK(q^.mutex)
      return index
    UNLOCK(q^.mutex)
    (back off exponentially and poll)
```

Figure 2. Generic mutual exclusion algorithm.

which holds the service number (RPC function) and arguments. Bulk data transfers use both queues—the *bulk data queue* holds the block of data. The queue structures have been carefully laid out to reduce cache-coherence traffic. Each packet occupies a distinct L2 cache line to avoid false sharing, and bulk data blocks begin on cache boundaries to increase copying speed.

Entries in the packet queue also contain a *type* and a *bulk index*. The type differentiates between short messages and bulk data transfers. It also serves as the handshake state in transferring data from a sender to a recipient. A claimed flag serves the latter purpose for the bulk data queue. The bulk index records the association between a bulk data transfer packet and the data itself. The bulk data queue is significantly shorter than the packet queue, allowing the queue block to fit into a reasonable amount of memory (293 kB).

3. Concurrent Algorithms

This section describes the six concurrent access algorithms that we evaluate in our study. Five are locking algorithms, with four drawn from the literature [18] and the fifth being the Solaris implementation of Posix mutexes. The last is our lock-free algorithm. We chose to implement no non-blocking algorithms, in part due to the difficulty of resolving storage issues and in part due to the added overhead inherent to queues based on pointers rather than on data packets. Also, the use of a request-response communication paradigm makes true non-blocking behavior moot, since an endpoint that fails to respond blocks application progress.

Each algorithm relies on either T&S or CAS, which are atomic with respect to all other memory accesses. TEST&SET(*address*) sets the value at an address to LOCKED and returns the previous value at the address. COMPARE&SWAP(*address, old, new*) compares the value at an address with an expected value, old. If the two values are equal, the operation writes a third value, new, into the address and returns TRUE. Otherwise, CAS returns FALSE.

The amount of cache traffic generated by a locking algorithm serves as the primary metric for abstract evaluation. Precluding starvation and providing fair access are also attractive qualities. The prevalence of universal primitives on modern machines has a subtle drawback for the latter issues, however. As with the simpler non-universal primitives like T&S, the number of times that a process can fail a CAS operation is not generally bounded by hardware. Hence no algorithm that makes use of CAS to manage centralized control information can eliminate the possibility of starvation. While such a scenario may seem unlikely with existing machines, it may become more common as the number of processors in an SMP grows.

The algorithms all use a common approach to message delivery. To enqueue a short message, a sender claims a packet in the destination queue structure with CLAIMPACKET (changing the type of the returned packet from FREE to CLAIMED), then fills in the packet. Claiming a packet involves concurrent access to the queue, but only a single sender accesses a packet while it is being filled. Once a packet is full, the sender changes its type to READY. This three-state handshake shortens the critical section by extracting packet-filling. For bulk data transfers, a sender uses CLAIMBULK to claim both a packet and a bulk data block, fills in both, and changes the packet type to READY-BULK. When a message at the head of a packet queue is ready for delivery, the receiver's poll operation advances the packet queue head and passes the arguments and, for bulk data transfers, the associated data block, to the service routine named by the packet. After this call returns, both the packet and the data block are marked as FREE.

3.1. Locking algorithms

Pseudo-code for claiming a packet with any of the locking algorithms appears in Figure 2. The algorithms differ only in their implementation of the LOCK and UNLOCK operations. Under the protection of a queue's lock, CLAIMPACKET checks for a free packet at the tail of the queue. When available, the packet is claimed, the queue tail is advanced, and the packet is returned. The tail packet may already be claimed—when the queue is full, for example—and, in this case, the operation stalls until it becomes available. CLAIMBULK has the same form as CLAIMPACKET but operates on both the packet and bulk data queues inside the critical section. CLAIMBULK also records the index of the associated data block in the packet.

When stalled on a full queue, both claim operations poll for incoming messages to avoid deadlock. Consider a group of endpoints with full request queues. If these endpoints simultaneously perform RPC's in a cyclic fashion, none can immediately succeed because the queues are full. Polling during backoff creates space in the queues and breaks the deadlock.

The first locking algorithm, Test & Set, uses a simple

```
CLAIMPACKET(^q)
    repeat
        index ← q^.tail
        next ← (index + 1) mod Q_LENGTH
    until COMPARE&SWAP(q^.tail, index, next)
    while TRUE
        if COMPARE&SWAP(q^.packet[index].type,
                        FREE, CLAIMED)
            return index
        (back off exponentially and poll)
```

Figure 3. Lock-free algorithm.

T&S lock. The second, Test & Test & Set, waits until a lock is released before making another attempt to obtain it, thereby reducing the amount of cache-coherence traffic generated while a lock is held.

Third, the Ticket Lock [20], further reduces cache-coherence traffic by ordering the processes waiting on a lock. To obtain a lock, a process obtains a ticket and waits for a service counter to show its ticket number. The ticket counter is incremented atomically to ensure that each process receives a different ticket. When a process releases a lock, it increments the service counter to allow the next process waiting on the lock to proceed. We found that placing both counters on a single cache line results in slightly better performance, and we report measurements only for that layout. The Ticket Lock exemplifies the CAS starvation phenomena on modern machines. Given hardware support for FETCH&ADD, the Ticket lock precludes starvation and is strictly fair [18]. Our implementation uses a FETCH&ADD operator built from CAS and hence cannot prevent starvation. It is fair only to processes that successfully obtain tickets.

Fourth is the Anderson Lock [1], which improves on the Ticket Lock by dividing the service counter into a separate flag for each process waiting on the lock. The lock operation obtains a slot assignment with CAS, then waits for the assigned slot to contain a lock indicator. When releasing a lock, a process moves the lock indicator from its slot into the next. The maximum number of processes must be known in advance to use the Anderson lock.

These four algorithms, together known as *spin locks*, have several drawbacks in practice. As the name implies, spin locks spin in a tight loop while waiting for a lock, potentially wasting valuable cycles. Spin locks also interact poorly with the operating system scheduler and admit deadlock when used with preemptive threads. Numerous preemption-safe locking solutions exist [19] and are supported in modern thread packages through integration with the operating system. The Solaris implementation of Posix mutexes, our final locking algorithm, exemplifies these solutions. A process that tries to obtain a lock held by another process enqueues itself on the lock and gives up its processor. Priority inversion is used to allow low priority processes to complete critical sections while high priority processes wait, and locks are implicitly reclaimed after abnormal termination. Unfortunately, the same close coupling with the operating system results in performance disadvantages on dedicated systems.

3.2. Lock-Free algorithm

Our lock-free algorithm is more robust than spin locks to preemption, yet avoids the high overhead inherent to operating system support. Pseudo-code for the algorithm appears in Figure 3. The analogue of the critical section in CLAIMPACKET consists of two steps. First, a sender obtains a packet assignment by atomically incrementing the queue tail using CAS. Next, the sender claims the assigned packet by changing its type from FREE to CLAIMED, again using CAS. The number of assigned packets may exceed the queue size, in which case multiple senders compete for a single packet in the second step. As with the locking algorithms, the lock-free algorithm avoids deadlock by polling when a packet claim fails.

For bulk data transfers, a sender uses a similar approach to claim a bulk data block before claiming a packet. The order defined by CLAIMBULK avoids a second deadlock scenario. Consider a process performing a bulk data transfer and holding the packet at the head of the packet queue. If the bulk data queue is full, the process cannot complete its operation, but neither can the receiving process receive any messages holding bulk data blocks until it receives the message at the head of the packet queue. By reserving space in the bulk data queue before competing for space in the packet queue, the algorithm avoids this situation.

Careful scrutiny of the lock-free algorithm allows us to predict its performance relative to the locking algorithms for both high and low levels of contention. The key to both predictions is the separation between assigning a packet and claiming it. This separation increases the number of synchronization primitives performed and results in slightly worse performance in the absence of contention. The same separation, however, results in a much shorter window of vulnerability to contention. The bottleneck step is packet assignment, but the CAS in this step is vulnerable to failure only between the completion of the queue tail load and the execution of the CAS, a period covering roughly a tenth of a microsecond on the Enterprise. The critical section in the locking algorithms spans at least two cache misses (the queue tail and the packet type) and totals at least a microsecond. Hence we expect the lock-free algorithm to outperform the locking algorithms under contention.

4. Performance Comparison

We now study performance over a range of access contention using tests from the message-passing literature.

	Test & Set	Test & Test & Set	Ticket Lock	Anderson Lock	Posix Mutex	Lock-Free
Latency (L)	-0.1	-0.1	-0.1	-0.2	0	-0.3
Send Overhead (o_s)	1.6	1.6	1.7	1.7	2.2	1.9
Receive Overhead (o_r)	1.2	1.2	1.2	1.2	1.3	1.2
Gap (g)	2.9	2.9	2.9	3.0	3.5	3.1
Round Trip Time (RTT)	5.5	5.4	5.5	5.5	6.9	5.6

Table 1. LogP parameters and round trip times in microseconds.

	Test & Set	Test & Test & Set	Ticket Lock	Anderson Lock	Posix Mutex	Lock-Free
1 writer	2.98	3.01	2.96	3.05	3.54	3.19
3 writers	3.03	3.46	3.18	3.22	10.00	3.08
7 writers	4.37	3.83	3.50	3.75	14.10	3.03

Table 2. Communication stress test times in seconds.

4.1. LogP microbenchmarks

We first measure point-to-point communication performance in terms of LogP. Assuming a small, fixed message length, the LogP model [7] characterizes communication networks as a set of four parameters: L, an upper bound on the network latency (wire time) between processors; o, the processor busy-time required to inject a message into the network or to pull one out; g, the minimum time between message injections for large numbers of messages; and P, the number of processors. The overhead o is often separated into send overhead, o_s, and receive overhead, o_r.

LogP parameters were measured using a microbenchmark from the suite described in [8] and appear in Table 1. The test uses one process as an RPC server and a second as a client and illustrates performance in the absence of contention. The negative latency values indicate overlap in time between the send and receive overheads [14], in this instance due to the poll operation. With the exception of the Posix mutex, the various algorithms are nearly equivalent, with send overhead and gap rising slightly as the complexity of the algorithm increases. Interacting with the operating system makes the Posix mutex performance significantly worse (more than 25%) than the others. Peak bandwidth using bulk transfers has less variation: the algorithms range from about 158 MB/s (Posix mutexes) to 166 MB/s (lock-free), or roughly 80% of the memory copy rate.

4.2. Communication stress test

We next observe the performance of many-to-one communication, a difficult if somewhat unlikely scenario for most message-passing systems. Using one process per physical processor, the test uses all but one process to write a total of one million integers to the remaining process. The results for up to eight processors appear in Table 2. This communication stress test generates the highest contention observable by an application using our Active Message layer, and may also be representative of more reasonable workloads on larger machines.

As apparent from the results, the test with a single writer is equivalent to the LogP gap measurement. With three and seven writers, the processes contend for access, and performance degrades for the locking algorithms. Surprisingly, the performance of the lock-free algorithm improves with more writers, allowing it to demonstrate the best performance under contention, where such approaches are traditionally expected to suffer. The shorter window of vulnerability in the lock-free algorithm accounts for the reduced impact of contention, but the improvements arise from a gap reduction effect found in many message-passing systems. The overhead of a poll operation is amortized over all messages accepted by that poll, making the gap for many-to-one communication lower than that for one-to-one communication. For our lock-free algorithm, gap reduction dominates the small increase in execution time due to contention.

4.3. Application analysis

We have measured performance at the extremes of contention. Extra complexity and multiple synchronization primitives add overhead at low levels of contention, but can improve performance when contention is high. The effect of this tradeoff at the application level is unclear, however. Applications are the natural metric for message-passing, since they use intrinsically interesting communication patterns. We now present execution times for three applications drawn from the Split-C application suite [6]. These programs are written in a bulk synchronous style—processors proceed through a sequence of coarse-grained phases, performing a global synchronization between each phase. Table 3 lists the input parameters and total memory requirement for each application run.

3-D FFT performs a fast Fourier transform in three di-

	Input Parameters	Memory
3-D FFT	256x256x256 values	328 MB
CON/comm	3D underlying lattice 512,000 nodes/processor 25% edges present	256 MB
CON/comp	2D underlying lattice 640,000 nodes/processor 40% edges present	320 MB
SAMPLE	262,144 nodes/processor	72.8 MB

Table 3. Application run input parameters.

mensions and typifies regular applications that rely primarily on bulk communication. The communication pattern is all-to-all, but is scheduled into many one-to-one phases to improve performance [3]. CON finds the connected components of a distributed graph. CON performs a large amount of fine-grained communication in a statistically well-defined pattern. The balance between computation and communication depends strongly on the input parameters. We selected both a communication-bound run and a second, computation-bound run. The input parameters for the first run result in a period of high contention and load imbalance near the end of the execution. SAMPLE sorts 32-bit integers using sample sort and represents applications that perform fine-grained, all-to-all communication.

Execution times for each run appear in Table 4. Our lock-free algorithm generally demonstrates the best performance for applications. The shorter window of vulnerability outweighs the cost of the second CAS. The exception is 3-D FFT, which uses primarily bulk communication and hence requires four CAS instructions for the lock-free algorithm. Despite this extra synchronization overhead, only the Ticket Lock is able to obtain better performance. Indeed, among the locking algorithms, the Ticket Lock provides the best performance for most applications, striking the proper balance between extra overhead and reduced cache traffic. As was evident from the microbenchmarks, the Posix mutex algorithm is not competitive. Binding each process to a separate physical processor improves Posix mutex performance by as much as a factor of two, but the cost of operating system support remains prohibitively expensive. Only the lock-free algorithm is both fast and robust to multiprogramming.

5. Related Work

The locking algorithms used in our work are drawn from a survey by Mellor-Crummey and Scott [18]. Herlihy is a good source for the theory of non-blocking behavior as well as some general practical approaches [10]. More practical implementations of non-blocking data structures are also abundant. Michael and Scott survey a variety of such algorithms and evaluate their performance on an SGI Challenge [19]. Massalin and Pu provide evidence that non-blocking approaches can be efficient when tailored to operating system data structures [17], and Greenwald and Cheriton show that more general approaches can also be successful in this regime [9]. Both kernel implementations make use of the double-compare-and-swap (DCAS) instruction, which performs simultaneous CAS operations on two independent words and is not generally available.

Valois differentiates between non-blocking and lock-free algorithms. The lock-free, array-based queue algorithm presented in [22] is similar to ours but requires significantly more complex operations and unaligned CAS instructions, making it "infeasible on a real machine."

The idea of passing messages through shared memory is not novel. Numerous applications make use of these techniques, as do a number of operating systems. However, these systems often avoid concurrent access by creating separate data structures for each sender-receiver pair. Cheriton and Kutter touch briefly on shared message segments, but only to mention their use in establishing a private segment for client-server communication [5]. Byrd builds optimistic high-level models of non-concurrent shared memory communication to aid in the design of a system that minimizes end-to-end latency [4]. A study by Lim et. al. [13] uses shared memory to route network traffic between SMP's through a proxy process, but again the data structures are duplicated for each communicating pair. Separate queues deliver superior results for one-to-one communication, but can be detrimental for more complex communication patterns. Polling each additional queue incurs a significant fraction of total message overhead in user-level communication layers [15].

Both Brewer et. al. [2] and Karamcheti and Chien [12] address concurrent message queues on the Cray T3D, a NUMA machine, with algorithms very similar to ours. Their algorithms use remote FETCH&INCREMENT support to claim queue entries from a static queue, but rely on the receiver to reset the queue after all entries have been claimed and processed. We avoid this boundary condition through the use of an additional synchronization primitive.

Although we used multiple processes to explore performance, our results also apply to the use of multiple threads. The main difference between these models lies in the need for bulk data transfers, since a thread can "send" a block of data by passing a pointer to the block in a short message. Karamcheti and Chien [12] describe an interesting "pull-based" algorithm using dynamic allocation and a link-based queue in the context of the T3D.

6. Conclusion

We have explored the performance of a variety of algorithms for managing concurrent access to message-passing queues and have demonstrated improved performance using a new lock-free algorithm that couples synchronization tightly to the data structure. The lock-free algorithm improves performance by reducing the window of vulner-

	Test & Set	Test & Test & Set	Ticket Lock	Anderson Lock	Posix Mutex	Lock-Free
3-D FFT	8.9	8.8	**8.6**	8.9	9.0	8.7
CON/comm	2.04	2.02	1.98	2.20	3.5	**1.96**
CON/comp	1.43	1.43	1.43	1.45	1.6	**1.40**
SAMPLE	2.3	2.3	2.5	2.8	9.5	**2.2**

Table 4. Application execution times in seconds (best performance in boldface).

ability to contention. Breaking the critical section into multiple steps requires the use of multiple synchronization primitives and adds some overhead in the absence of contention, but the cost of synchronization primitives on modern hardware is fairly low—about fifteen cycles on the Enterprise 5000—and the benefits of reduced contention dominate the behavior of the algorithm for applications.

Fast shared memory message-passing is an important problem for SMP's and for clusters of such machines, but few studies have considered the merits of concurrent access algorithms for this domain. We have applied approaches from non-blocking theory to create a fast, lock-free solution. Our algorithm provides a greater degree of robustness to multiprogramming and demonstrates performance superior to traditional locking techniques for real applications. In the process, we have demonstrated that lock-free techniques can be effectively applied to a specific, practical problem. Other interesting problems might also benefit from a reexamination of the concurrent access literature.

References

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

[2] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. D. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Symposium on Parallel Algorithms and Architectures*, 1995.

[3] E. A. Brewer and B. C. Kuszmaul. How to Get Good Performance from the CM-5 Data Network. In *International Parallel Processing Symposium*, Apr. 1994.

[4] G. T. Byrd. Models of Communication Latency in Shared Memory Multiprocessors. Tech. Report CSL-TR-93-596, Stanford University, Dec. 1993.

[5] D. R. Cheriton and R. A. Kutter. Optimized Memory-Based Messaging: Leveraging the Memory System for High-Performance Communication. *Computing Systems*, 9(3):179–215, 1996.

[6] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Supercomputing*, pp. 262–73, Nov. 1993.

[7] 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. In *Principles and Practice of Par. Programming*, May 1993.

[8] D. E. Culler, L. T. Liu, R. P. Martin, and C. O. Yoshikawa. Assessing Fast Network Interfaces. *IEEE Micro*, 16(1):35–43, Feb. 1996.

[9] M. Greenwald and D. Cheriton. The Synergy between Non-Blocking Synchronization and Operating System Structure. In *Operating Systems Design and Impl.*, Oct. 1996.

[10] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. Tech. Report CRL 91/10, DEC CRL, Oct. 1991.

[11] M. P. Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. In *Symposium on Principles of Distributed Computing*, Aug. 1988.

[12] V. Karamcheti and A. A. Chien. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. In *Int. Symp. on Comp. Arch.*, pp. 298–307, June 1995.

[13] B.-H. Lim, P. Heidelberger, P. Pattnaik, and M. Snir. Message Proxies for Efficient, Protected Communication on SMP Clusters. Tech. Report #RC 20522 (90972), IBM Almaden, Aug. 1996.

[14] L. T. Liu and D. E. Culler. Evaluation of the Intel Paragon on Active Message Communication. In *Intel Supercomputer Users Group Conference*, Jun. 1995.

[15] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *SC97: High Performance Networking and Computing*, Nov. 1997.

[16] A. M. Mainwaring and D. E. Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Tech. Report #CSD-96-918, U. C. Berkeley, Oct. 1996.

[17] H. Massalin and C. Pu. A Lock-free Multiprocessor OS Kernel. Tech. Report CUCS-005-91, Columbia University, Jun. 1991.

[18] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. on Comp. Systems*, 9(1):21–65, Feb. 1991.

[19] M. M. Michael and M. L. Scott. Relative Performance of Preemption-Safe Locking and Non-Blocking Synchronization on Multiprogrammed Shared Memory Multiprocessors. In *International Parallel Processing Symposium*, 1997.

[20] D. P. Reed and R. K. Kanodia. Synchronization with Eventcounts and Sequencers. *Communications of the ACM*, 22(2):115–23, Feb. 1979.

[21] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. In *Hot Interconnects IV*, pp. 41–52, Aug. 1996.

[22] J. D. Valois. Implementing Lock-Free Queues. In *International Conf. on Par. and Dist. Computing Sys.*, Oct. 1994.