A Performance Comparison of Fast Distributed Synchronization Algorithms

Theodore Johnson
Dept. of CIS, University of Florida
Gainesville, Fl 32611-2024
ted@cis.ufl.edu

Abstract

Distributed synchronization is an essential component of parallel and distributed computing. Several fast and low-overhead distributed synchronization algorithms have been proposed. Each of these algorithms required $O(\log n)$ messages per critical section entry and $O(\log n)$ bits of storage per processor. Asymptotic performance estimates do not proclaim any of the algorithms to be a winner, but no performance comparison of the algorithms has yet been performed. In this paper, we make a comparative performance study of four distributed semaphore algorithms. Each of these algorithms represents a different approach to maintaining distributed information. Since the algorithms we study are the basis for distributed synchronization, distributed virtual memory, coherent caches, and distributed object systems, our results have implications about the best methods for their implementation. We find that the distributed synchronization algorithm of Chang, Singhal, and Liu has the overall best performance, though other algorithms are more efficient in special cases. In a system of 350 processors, the CSL algorithm requires only six messages per critical section entry, including the initial request and the token response messages.

Keywords: Distributed Synchronization, Performance Analysis, Distributed Algorithms, Distributed Virtual Memory

1 Introduction

Distributed synchronization is an important activity that is required to coordinate access to shared resources in a distributed system. A set of n processors synchronize their access to a shared resource by requesting an exclusive privilege to access the resource. The privilege is often represented as a *token*. Access to the token can represent the ownership of a page of distributed shared memory, exclusive permission to update shared data, permission to access an external resource, and so on. The processors synchronize by sending and interpreting messages according to a synchronization protocol. We assume that every message that is sent is eventually received. Because distributed synchronization is such a fundamental

activity, much work has been done to develop efficient algorithms. However, little work has been done to compare the performance of distributed synchronization algorithms

In this paper, we discuss four fast and low overhead distributed synchronization algorithms. Three of the algorithms has previously been presented in the literature, and the fourth is novel with this paper. Each of the algorithms require $O(\log n)$ bits of storage per processor (the $O(\log n)$ bits are required to store the names of $O(1)$ processors), and $O(\log n)$ messages per critical section entry. The low space and message passing overhead make them scalable and practical for implementation. Since each of the algorithms has the same asymptotic complexity, a performance analysis is required to determine which is the best under different circumstances.

We present a simulation study of the four algorithms. We examine the number of messages per critical section entry and the time to pass the token under a variety of loadings and numbers of processors. We conclude that the algorithm by Chang et al. is the best overall, but that other algorithms are better is special cases. For example, Raymond's algorithm is the best under a heavy load and the algorithm new to this work is best when the load on the critical section is close to 100%. We find that distributed synchronization is very efficient. With 350 processors, distributed synchronization requires as little as four messages per critical section entry under a 100% load, and six messages per critical section entry under a lighter load. Since this includes the message that requests the token and the message that releases the token there are only two to four overhead messages.

1.1 Background Work

Considerable attention has been paid to the problem of distributed synchronization. Lamport [6] proposes a timestamp-based distributed synchronization algorithm. A processor broadcasts its request for the token to all of the other processors, which reply with a permission. A processor implicitly receives the token when it receives permissions from all other processors. Ricart and Agrawala [12] and Carvalho and Roucairol [1] improve on Lamport's algorithm by reducing the message passing overhead. However, all of these algorithms require $O(n)$ messages per request.

Thomas [14] introduces the idea of *quorum consensus* for distributed synchronization. When a processor requests the token, it sends a vote request to all of the other processors in the system. A processor will vote for the critical section entry of at most one processor at a time. When a processor receives a majority of the votes, it implicitly receives the token. The number of votes that are required to obtain the token can be reduced by observing that the only requirement for mutual exclusion is that any pair of processors require a vote from the same processor. Maekawa [8] presents an algorithm that requires $O(\sqrt{n})$ messages per request and $O(\sqrt{n}\log n)$ space per processor. Kumar [5] presents the hierarchical quorum consensus protocol, which requires $O(n^{.63})$ votes for consensus, but is more fault tolerant than Maekawa's algorithm.

Li and Hudak [7] present a distributed synchronization algorithm to enforce coherence in a distributed shared virtual memory (DSVM) system. In DSVM, a page of memory in a processor is treated as a cached version of a globally shared memory page. Typical cache coherence algorithms require a home site for the shared page, which tracks the positions of the

copies of the page. The 'distributed dynamic' algorithm of Li and Hudak removes the need for a fixed reference point that will locate a shared page. Instead, every processor associates a pointer with each globally shared page. This pointer is a guess about the current location of the page. When the system is quiescent, the pointers form a tree that is rooted at the current page owner. The tree is kept short by using path compression, which guarantees an amortized $O(\log n)$ bound on the number of messages per request. Trehel and Naimi [16, 15] present two algorithms for distributed mutual exclusion that are similar to the 'distributed dynamic' algorithm of Li and Hudak. Chang, Singhal, and Liu [2] present an improvement to the Trehel and Naimi algorithm [15] that reduces the average number of messages required per critical section entry.

Raymond [11] has proposed a simple synchronization algorithm that can be configured to require $O(\log n)$ storage per processor and $O(\log n)$ messages per critical section request. The algorithm organizes the participating processors in a fixed tree. Each processor points to the neighbor that lies on the path to the token holder. Neilsen and Mizuno [10] present an improved version of Raymond's algorithm that requires fewer messages because it passes tokens directly between processors instead of through the tree. Woo and Newman-Wolfe [17] use a fixed tree based on a Huffman code.

Some shared memory synchronization algorithms can be easily modified to construct distributed synchronization algorithms. These algorithms include the MCS contention-free lock [9] and cache coherence protocols such as the Scalable Coherent Interface (SCI) [3]. However, these algorithms require the use of a centralized lock manager.

Only a little work has been done to make a performance study of distributed synchronization algorithms. Ricart and Agrawala [13] make a simulation study of an $O(n)$ message passing algorithm. Chang, Singhal, and Liu [2] use a simulation study to show that their improvements to Trehel and Naimi's algorithm actually does result in better performance.

2 The Algorithms

All four algorithms that we study in this paper require $O(\log n)$ messages per critical section entry, and $O(\log n)$ bits per processor. In addition, the algorithms are symmetric: there is no centralized processor that performs a special function. The symmetric protocol rules out techniques such as the MCS-lock or the SCI protocol. While these protocols require $O(1)$ messages per critical section entry, they are not fault tolerant and they make a single processor perform most of the work, even if the processor might not require the lock.

Since each processor is allowed only $O(\log n)$ space, it can remember information about $O(1)$ other processors. The space restriction makes the lock scalable. However, it disallows the possibility of storing a map of the state of the other processors in the system. Similarly, the $O(\log n)$ message passing restriction does not allow many processors to be told about changes in the state of the system. An algorithm that lets any processor find the lock holder at must in general use a hierarchical structure (i.e., a tree) to guide the protocol.

There are two approaches to maintaining a tree that points to the token holder: the fixed-tree approach and the path-compression approach. In the fixed tree approach, a tree structure is imposed on the processes, and processes are generally restricted to communicating with their neighbors in the tree. Each processor stores the direction in the tree where the

token resides. In the path-compression approach, each processor stores a best guess about which processor holds the token. Following a sequence of guesses leads to the token holder. When a processor processes a request for the token, it changes its guess about the token holder to be the requester. As a result, if any request for the token follows a long chain of guesses, the tree is *compressed*.

We consider four algorithms, three of which have previously been published in the literature (Raymond's algorithm, Chang, Singhal, and Liu's algorithm, and Neilsen and Mizuno's algorithm), and a fourth which is original with this paper (which we call *List Lock*). The algorithms use a variety of techniques for maintaining information about which process holds the token and about which processes are waiting to use the token. We give only brief descriptions of the algorithms here. We provide citations that give further details about the algorithm implementations.

2.1 Raymond's Algorithm

Raymond [11] proposes an algorithm for maintaining a distributed lock which makes use of a tree structure that is imposed on the processes. Each processor keeps a pointer, `dir`, to the neighbor which is the root of the subtree where the token is located (see Figure 1). In addition, each processor keeps a FIFO queue of pending requests. The possible entries of the queue are the processor itself and the processor's neighbors. When a processor that does not hold the token receives a request for the token (perhaps generated locally), it puts the request into the queue. If the queue was previously empty, it forwards the request in the direction of the token holder. When the processor receives the token, it removes the entry at the head of the queue. If the processor itself was at the head of the queue, the processor enters the critical section. Otherwise, the processor forwards the token to the neighbor which was at the head of the queue, and then sends a request to the neighbor. When the token holder receives a request, it either stores the request in the queue (if it is in the critical section) or it replies with the token.

The execution of Raymond's protocol is illustrated in Figure 2. A request is indicated the by name of the requester in parentheses, and the FIFO queue is indicated by a box attached to the processor. In Figure 2, D requests the token. Since D's FIFO was empty, the request is forwarded to B. B puts D into its FIFO and forwards the request to A. Notice that the request sent to A is in B's name. Later, C requests the token. Since B's FIFO is not empty, it has already requested the token. So, C's request is put into B's FIFO and no further action is taken. Notice that the return path for the token (leading to processor D) is stored in the FIFOs of each processor.

One property of Raymond's algorithm is that the number of messages required for synchronization decreases as the request activity increases. If your request reaches a node that has already processed a request, no further messages are sent. The chance of this happening increases as the number of waiting processors increases.

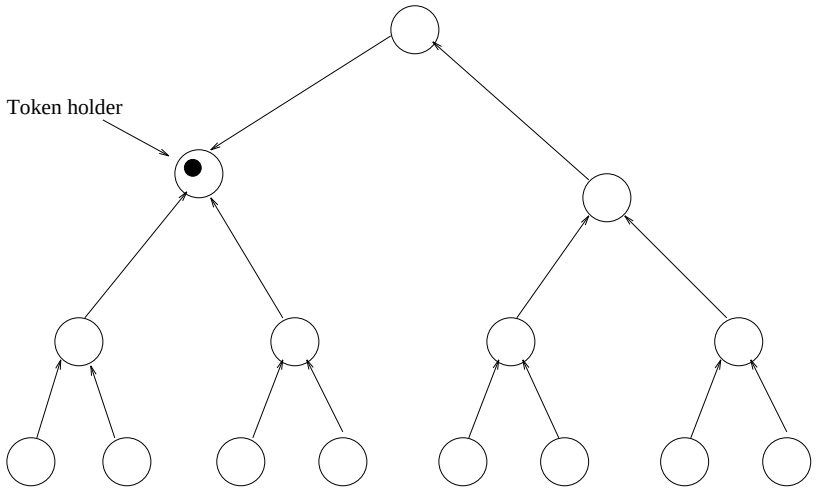

Figure 1: Process structure in Raymond's algorithm.

2.2 Neilsen and and Mizuno's Algorithm

One potential source of inefficiency in Raymond's algorithm is that the token must travel through the tree in order to reach the process which next accepts the token. Neilsen and Misuno [10] observed that it is possible to pass the token directly to the requester, since the identity of the original requester can be attached to the request. Neisen and Mizuno used this observation to develop an algorithm which we will call the NM algorithm.

The NM algorithm adds a dynamic waiting chain to the static tree to permit the token to be passed directly to the next processor in line. If a process holds or is requesting the token, it maintains a pointer, `next`, which points to the next processor in line for the token (or is `NIL` if there is no next processor). To support the waiting chain, the fixed tree no longer points to the token holder, but rather to the end of the chain (which is the token holder if there are no waiting processes). We call the tree pointer `dir`. The structure of the NM algorithm is illustrated in Figure 3. The `dir` pointers are solid arrows and the `next` pointers are dashed arrows. The nodes marked with an 'r' represent nodes that are waiting for the token.

When a processor that is not requesting the token receives a request, it forwards the request to `dir`, then sets `dir` to the neighbor that sent it the request. When a processor that is requesting the token receives a request, it checks the value of `next`. If `next` is `NIL`, it sets `next` to the requester (i.e., makes the requester next-in-line for the token). If `next` is not `NIL`, the request is forwarded to `dir` (i.e., sent to the end of the list). In either case, the requester is the processor's best guess about where is the end of the list, so `dir` is set to the neighbor who sent the request.

The execution of the NM algorithm is illustrated in Figure 4. The solid arrows represent the `dir` pointers and the dashed arrows represent the `next` pointers. Originally, C points to A (with `dir`) and A points to B. When C makes its request, it sends its request to A and sets `next` to `NIL`. We represent the edge (C,A) by a dotted line because it no longer exists at C after C sends its request. When A receives the request, it sets `dir` to C, then forwards the

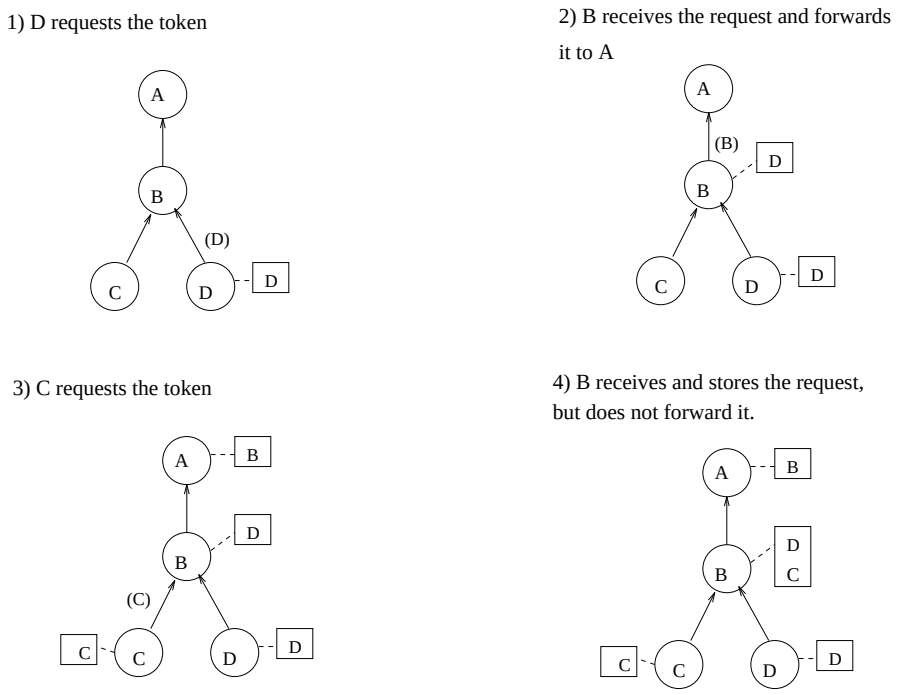

Figure 2: Sample execution of Raymond's algorithm.

request to B. Notice that unlike Raymond's algorithm, the name attached to the request is the name of the original requester. Finally, B receives C's request. Since B is itself waiting and is at the end of the list, B adds C to the list by setting `next` to C. Notice that C is now at the end of the list, and that the `dir` pointers now lead to C.

If the demand for the critical section is low, then the Neilsen-Mizuno algorithm should require about half the messages that Raymond's algorithm requires. In addition, there is less wasted time in passing the token because the token is sent directly to the next waiting process. However, when the demand for the token is high Raymond's algorithm can usually stop forwarding a request early while in the NM algorithm the request must search for the tail of the waiting chain. As a result, it is not clear which protocol is more efficient.

2.3 Chang, Singhal, and Liu's Algorithm

The algorithm of Chang Singhal and Liu [2] (which we call the CSL algorithm) makes aggressive use of path compression to achieve good performance. Their algorithm is based on the algorithm by Li and Hudak. Each processor maintains a guess about which processor holds the token. This guess is stored in the variable `dir`. If a processor that neither holds nor is requesting the token receives a request, it forwards the token to the processor indicated by `dir`, then sets `dir` to the name of the requesting processor. This process is illustrated in Figure 5. A's request must make four hops to reach the token, but subsequently requests form B, C, and D need to make only one hop.

A weakness of the algorithm proposed by Li and Hudak is that it requires every processor

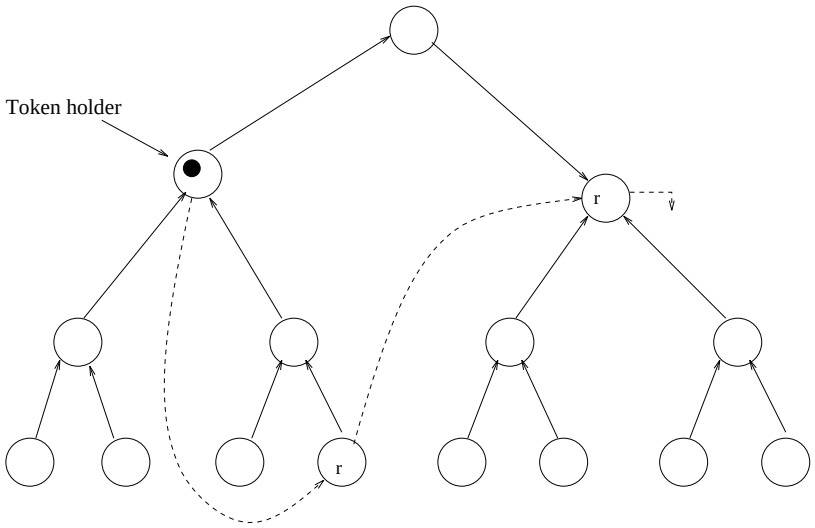

Figure 3: Process structure in the NM algorithm.

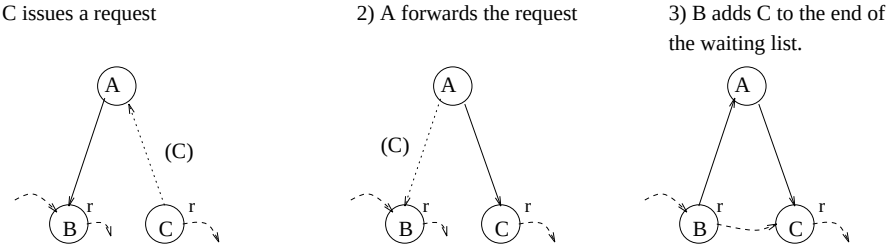

Figure 4: Sample execution of the NM algorithm.

to maintain $O(n)$ storage to hold the identities of the blocked processors. The CSL algorithm reduces the $O(n)$ space overhead to an $O(\log n)$ space overhead by storing the set of blocked processors in a distributed list (as in the NM algorithm).

When a processor requests the token, it sends a request message to the processor indicated by `dir`. It then sets an additional pointer, `next` to `NIL`. If a processor that holds or is waiting for the token receives a request, and its `next` pointer is `NIL`, it sets `next` to the identity of the processor that sent the request. Otherwise, it forwards the request to the processor indicated by `dir`, and sets `dir` to the requesting processor.

Among the processors that hold the token or are waiting, the `next` variable forms a queue of the blocked processors. If a processor is waiting and its `next` pointer is `NIL`, the processor is (effectively) at the end of the waiting queue. If `next` is not `NIL`, the end of the waiting queue is at the processor pointer to by `dir`, or beyond. Since the requesting processor will become the one at the end of the list, it is appropriate to set `dir` to the identity of the requesting processor. When the token holder releases the token, it sends the token to `next` if `next` is not `NIL`. Otherwise, the token holder keeps the token without using it.

The structure of the CSL algorithm is shown in Figure 6. The solid arrows represent the `dir` pointers, and the dashed arrows represent the `next` pointers. The processors that are

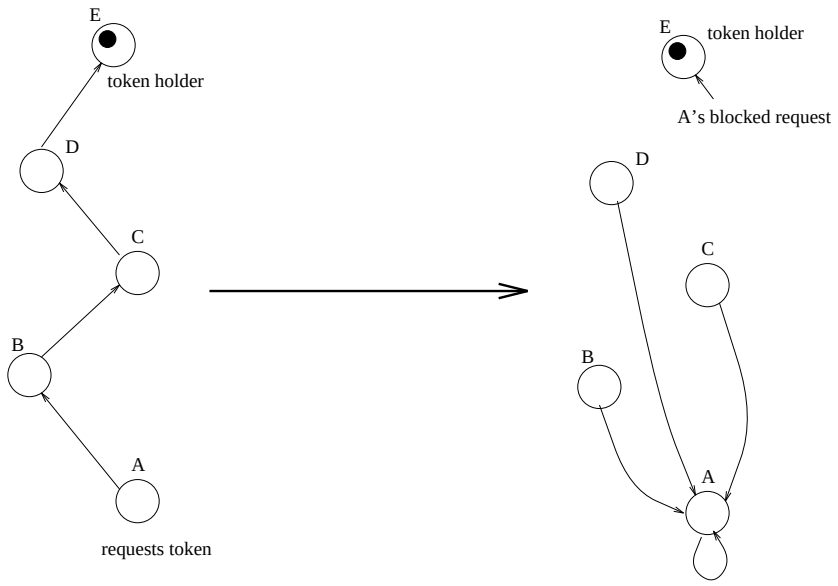

Figure 5: An example of path compression.

not requesting the token lie on a path that leads either to the token holder or to a processor that is in the waiting list. The `next` pointers form a list of blocked processes whose head is the token holder. In addition, the `dir` pointers in the list of blocked processors point to another blocked processor that is closer to the end of the list.

A sample execution is shown in Figure 7. The solid lines represent the `dir` pointers, the dashed lines represent the `next` pointers, and the dotted lines represent the direction that message travels when the corresponding pointer has already been erased. First, processor D sends its request to A (where D's value of `dir` used to point). A sets its value of `dir` to D and forwards the request to B. Similarly, B modifies its value of `dir` and forwards the request to C. C is at the end of the waiting list, so C sets its value of `next` to D. Notice that path compression operates on waiting as well as non-waiting processors. As a result, waiting processors have a good guess about the end of the waiting list and the CSL algorithm has good performance in practice.

2.4 List Lock Algorithm

In this section, we describe an algorithm, which we call the *List Lock*, that was inspired by our previous work on a distributed priority lock that used path compression to achieve good performance [4]. To transform a priority lock into a non-prioritized lock, we prioritize a request based on the time of the request (to ensure causality, we can use a Lamport timestamp).

In the prioritized synchronization algorithms, the processors that are neither requesting nor holding the token keep a guess, `dir` of a processor that either holds or is waiting for the token. When one of these processors receives a request, it forwards the request to `dir`, then sets `dir` to the identity of the requester. Processes that are waiting for the token form

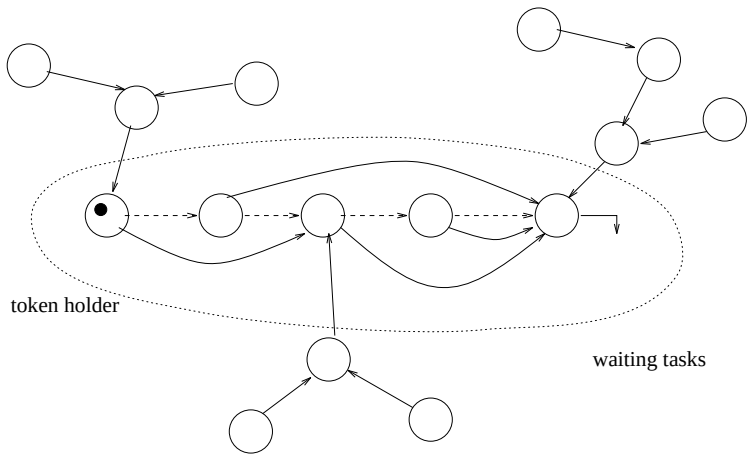

Figure 6: Processor structure in the CSL algorithm.

a ring using the variable `next`, ordered by the priority of their request. An exception is the lowest priority waiting processor, which points to the highest priority waiting processor (using the variable `next`) to allow a high priority request to find its position. The token holder is required to point to one of the processors in the ring of waiting processors.

When a waiting processors receives a request, it checks to see if the priority of the requesting processor is between the priority of its request and the priority of its successor's request. If so, the requesting processor is inserted into the waiting ring, and the requester is informed of its position in the ring. Otherwise, the request is sent to the successor in the ring.

To improve the performance of the algorithm, we can make the observation that we no longer need to satisfy requests in strict priority order, as long as we avoid starvation. If a processor receives a request that is of a higher priority than its own request, the requester is inserted after the receiving processor. Therefore, there is no need to maintain a ring. Instead, the waiting processors form a list, and the token holder points to the head of the list.

We make the further observation that if a request arrives at a waiting processor, and the priority of the request is not too much lower than the priority of the successor processor, the requester can be inserted between the receiving processor and the successor processor. In particular, the processor checks if the time of the previous entry of the requester is later than the request time of the successor. If so, the request is forwarded to the requester. Else the request is inserted between the receiver and the successor. If the set of processes that request the critical section is finite, then eventually every request is satisfied.

An example execution is shown in Figure 8. The actions taken by non-requesting processors is the same as in the CSL algorithm, so we look at the actions taken at the waiting list only. Processor A is requesting the token, and processors B, C, and D are in the waiting list. When B receives A's request, B compares A's last entry time to C's request time (A's last entry time is sent with the request, C's request time is stored with the `next` pointer).

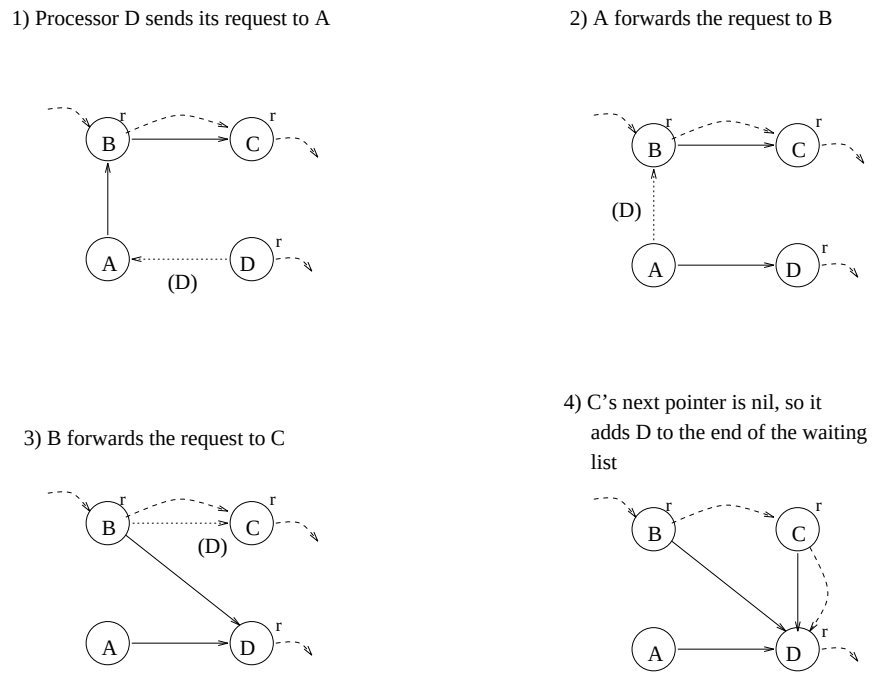

Figure 7: Example execution of the CSL algorithm.

Since A last entered after C requested, A's request is sent to C. C compares A's last entry time to D's request time, and finds that A can enter the list before D. C sets its `next` pointer to A, and records A's request time (which is carried with A's request). A is informed of its successor in the list, and sets its `next` pointer accordingly.

There are several implementational details that must be considered. For example, the `dir` variable does not have a meaningful value during the time interval between which a processor requests the critical section and when it is informed of its position in the waiting list. These details all can be handled, and we refer the interested reader to our previous technical report for the solution methods.[1]

By these and several other minor optimizations, we achieved an algorithm that has good performance. We note that while both the CSL and the list lock algorithm both use path compression, they take very different approaches to managing the list of waiting processors. The CSL algorithm attempts to forward a request to the end of the list, and also maintains a good estimate of the location of the end of the list. The list lock algorithm attempts to insert a request into the list immediately. It turns out that both methods have similar performance even under a heavy load when the waiting lists are long.

[1] **Note for the reviewer:** Space limitations prevent us from providing a full description of the algorithm. Since our performance results do not proclaim this algorithm to be the best, a full description is not the best use of space. We discuss the algorithm because its approach complements the approach of the other three algorithms.

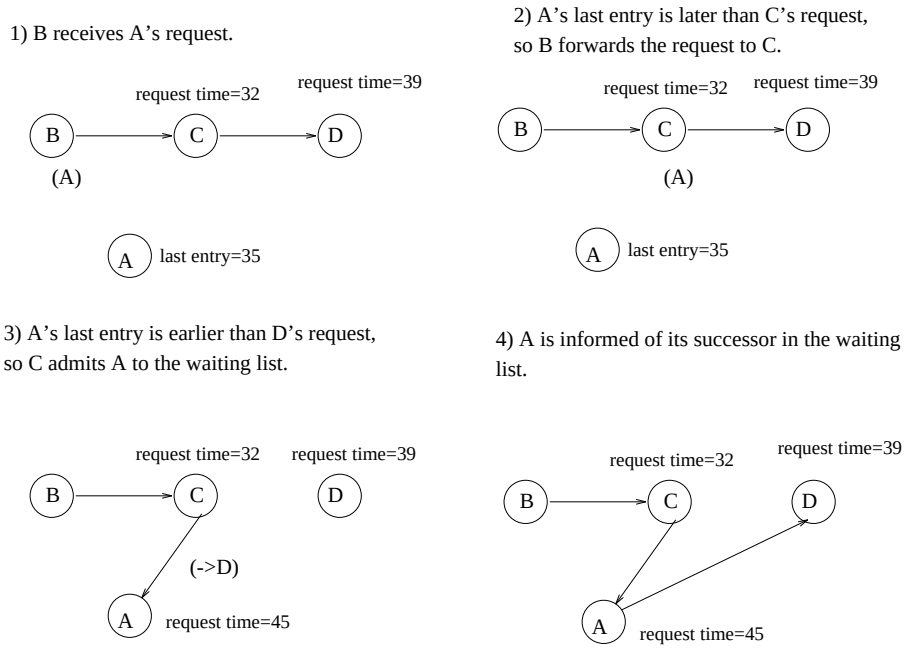

Figure 8: Example execution of the List Lock algorithm.

3 Performance

Since a theoretical analysis of the three distributed priority lock algorithms does not clearly show that one algorithm is better than another, we make a simulation study of the algorithms. The simulator modeled a set of processors that communicate through message passing. All delays are exponentially distributed. The parameters to the simulator are the number of processors, the message transit delay (mean value is 1 tick), the message processing delay (1 tick), the time between releasing the token and requesting it again (the inter-access time, varied), and the time that a token is held once acquired (the release delay, 10 ticks). The fixed-tree algorithm uses a nearly-complete binary tree (requiring about the same number of bits per processor as the CSL algorithm).

We wanted to investigate the influence of the request load on the performance of the algorithms. Since the different algorithms have different degrees of overhead, it is not meaningful to measure the lock utilization directly. Instead, we define the *load*, l, on the critical section to be $l = nC/R$, where n is the number of processors, C is the average critical section execution time, and R is the average time between releasing the critical section and requesting it again. If the algorithm overhead is negligible, then a load $l < 1$ will result in a $(100 * l)\%$ lock utilization. Since the requests are generated by a finite population, it is meaningful to have a load larger than 1. For example, a load of 200% means that on average half of the processors are waiting for the critical section.

We ran the simulator for varying numbers of processors and varying loads. For each run, we executed the simulation for 100,000 critical section entries. We collected a variety of statistics, but principally the amount of time to finish the simulation (which captures the

time overhead of running the protocol), and the number of messages sent.

In our first set of experiments, we plotted the number of messages sent per critical section entry for a 50%, a 75%, and a 100% load. The results are shown in Figures 9 through 11.

When the load is light (i.e., 50%), the CSL algorithm requires the fewest messages while Raymond's algorithm requires the most messages. When the load is heavy (i.e., 100%), it is Raymond's algorithm that requires the fewest messages. The wide swing in the number of messages per critical section entry that Raymond's algorithm requires is due to its ability to let a previous request subsume a current request. As the request load increases, it becomes more likely that a request will reach a node that has already processed a request, and so stop early. In Figure 10 (75% load), the number of messages required Raymond's algorithm drops sharply between 160 and 200 processors. This phenomenon occurs because the system is driven into a heavy load. We will return to this subject shortly.

The List Lock algorithm typically requires less than one more message than the CSL algorithm requires. Their dynamic performance is very similar in spite the different rules they use admit processes to the waiting list. The difference is primarily due to the fact that in the List Lock algorithm, a processor must be informed of its position in the waiting list requiring the one additional message. If the waiting list is very long (the load is high and the number of processes is large), the List Lock algorithm is slightly more efficient in admitting processes to the waiting list.

If the load is light the NM algorithm requires fewer messages than Raymond's algorithm, but more messages than the CSL algorithm or than the List Lock algorithm. Unlike Raymond's algorithm, the NM algorithm does not take advantage of previously established paths. As a result, its performance does not improve as the load increases and becomes the worst of the four algorithms.

Figures 9 through 11 show that the performance of the algorithms strongly depends of the request load. In Figures 12 through 14, we plot the number of messages required per critical section entry against the load, for 20, 120, and 350 processors.

These figures more clearly show the influence of the load on the performance of the algorithms. Raymond's algorithm improves significantly in a heavy load. In addition, the dramatic performance improvement occurs at a lower loading as the number of processors increases. Both the CSL and the List Lock algorithm require fewer messages as the load increases. Since a high load means that there are fewer non-waiting processes the length of the path to the waiting list decreases, accounting for most of the gain. Once a request reaches the waiting list, the List Lock algorithm admits the request to the list faster than the CSL algorithm does, and the List Lock algorithm overcomes its handicap of one additional overhead message. The NM algorithm is not affected by changes in the load.

We have observed that Raymond's algorithm appears to go into a heavy load condition when the load is 75% and the number of processors increases. This phenomenon occurs because the load on Raymond's algorithm does increase as the number of processors increases. Unlike the other three algorithms, Raymond's algorithm does not pass the token directly to the next token holder, instead the token must travel through a pre-specified return path. Therefore, as the number of processors increases, the length of the return path increases, and the effective critical section execution time (i.e., the minimum time between critical section entries) increases. In Figure 15, we plot the lock utilization (the percentage of time that

the lock is held or in transit) against the number of processors for a 75% load. The lock utilization for Raymond's algorithm increases quickly to 100%, while the lock utilization of the other three algorithms stays at about 87%. We observe that the drop in the number of messages required by Raymond's algorithm in Figure 10 corresponds to the point where the utilization hits 100%.

Since Raymond's algorithm increases the effective critical section execution time, we need to explore to what extent the additional overhead will reduce the throughput of a system that fully utilizes the lock. We plot the average amount of time between critical sections against the number of processors when the load is 100% in Figure 16. There is a slight decrease in the execution time as the number of processors increases from 10 to 40. This occurs because the idle time (caused by statistical fluctuations) decreases, and is almost zero with 80 processors. Surprisingly, as the number of processors increases, the time between critical sections does not increase in Raymond's algorithm. The reason is that Raymond's algorithm prefers to give the token to a process that is close to the requester instead of to a processor that is distant from the requester. As a result, the distance that the token travels stays about the same as the number of processors increases.

We finish by noting that some of the algorithms are "more fair" than others. While all of the algorithms guarantee that all waiting processes are served eventually, some of the algorithms prefer to pass the token to "close" processors. In Figure 17, we plot the maximum time between requesting the token and receiving the token as we vary the number of processors in a 100% load. The CSL and the NM algorithm are very fair, because they require processors to always join the end of the waiting list. Raymond's algorithm is somewhat less fair, since it tends to serve requests in one subtree before moving on to a different subtree. The List Lock algorithm is the least fair, because it lets processes cut into line. However, the differences in the maximum waiting time are related by a constant. All experience linear growth, with the slope of the waiting time for Raymond's algorithm about twice that of the CSL algorithm, and the slope of the waiting time of the List Lock algorithm is about six times that of the CSL algorithm.

4 Conclusions

We have presented a performance study of four fast and low overhead distributed synchronization algorithms. Our findings include:

1. Recently proposed distributed synchronization algorithms are as fast and efficient as advertised. In a system of 350 processors, only four to six messages are required per critical section entry. Since this counts the initial request and the receipt of the token, there is only a two to four message overhead.

2. The CSL algorithm is the best overall, as it is a simple algorithm, it is fair, it imposes a small overhead on the effective critical section execution time, and requires the fewest number of messages when the load is light (the expected case).

3. Raymond's algorithm is the best asymptotically in the following sense: Given a load on the critical section, increase the number of processors and count the number of

messages per critical section entry. Eventually Raymond's algorithm will be driven into a heavy load and will require the fewest messages among all of the algorithms. In addition, it will impose a bounded overhead on the effective time to execute the critical section.

4. The List Lock algorithm requires few messages (within one of the CSL algorithm), and requires fewer messages than the CSL algorithm when the load is high and the number of processors is large. However, it is a complex algorithm, and is the least fair of the algorithms that we considered.

5. Path compression algorithms generally perform better than fixed-structure algorithms. This study assumed uniform requests. If a small subset of processors generates most of the requests, path compression algorithms are even better than fixed-structure algorithms.

6. The technique of allowing processes to cut in line is more effective than requiring processes to always find the end of the line. In the setting considered here, the advantage was more than offset by the additional complexity of the algorithm.

References

[1] O.S.F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Comm. of the ACM*, 26(2):146–147, 1983.

[2] Y.I. Chang, M. Singhal, and M.T. Liu. An improved $O(\log(n))$ mutual exclusion algorithm for distribtued systems. In *Int'l Conf. on Parallel Processing*, pages III295–302, 1990.

[3] D.V. James, A.T. Laundrie, S. Gjessing, and G.S. Sohi. Scalable coherent interface. *Computer*, 23(6):74–77, 1990.

[4] T. Johnson and R. Newman-Wolfe. A comparison of fast and low overhead distributed priority locks. Technical report, U. Florida Dept. of CIS, 1994. available at ftp.cis.ufl.edu:/cis/tech-reports/tr94/tr94-022.ps.Z.

[5] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Trans. on Computers*, 40(9):994–1004, 1991.

[6] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[7] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, 1989.

[8] M. Maekawa. A sqrt(n) algorithm for mutual exlcusion in decentralized systems. *ACM Trans. on Computer Systems*, 3(2):145–159, 1985.

[9] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Computer Systems*, 9(1):21–65, 1991.

[10] M.L. Neilsen and M. Mizuno. A dag-based neilsen for distributed mutual exclusion. In *International Conference on Distributed Computer Systems*, pages 354–360, 1991.

[11] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. on Computer Systems*, 7(1):61–77, 1989.

[12] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Comm. of the ACM*, 24(1):9–17, 1981.

[13] G. Ricart and A.K. Agrawala. Performance of a distributed network mutual exclusion algorithm. Technical Report TR-774, U. Maryland College Park Department of Computer Science, 1991.

[14] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[15] M. Trehel and M. Naimi. A distributed algorithm for mutual exclusion based on data structures and fault tolerance. In *IEEE Phoenix Conference on Computers and Communications*, pages 36–39, 1987.

[16] M. Trehel and M. Naimi. An improvement of the log(n) distributed algorithm for mutual exclusion. In *Proc. IEEE Intl. Conf. on Distributed Computer Systems*, pages 371–375, 1987.

[17] T.K. Woo and R. Newman-Wolfe. Huffman trees as a basis for a dynamic mutual exclusion algorithm for distributed systems. In *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems*, pages 126–133, 1992.

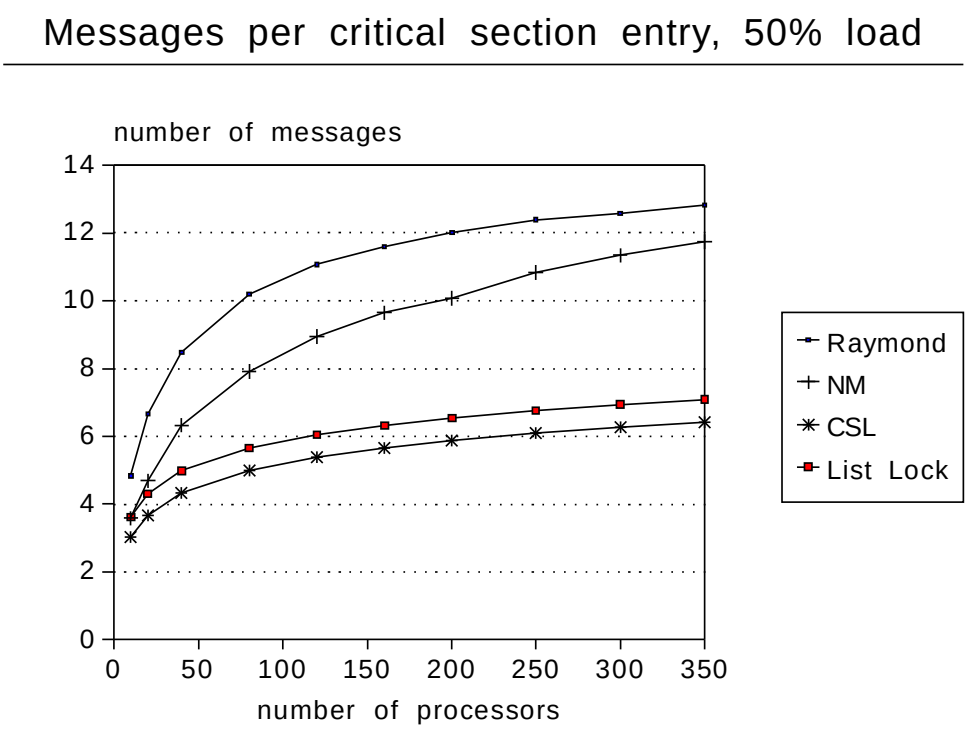

Figure 9: Number of messages per critical section entry, 50% load.

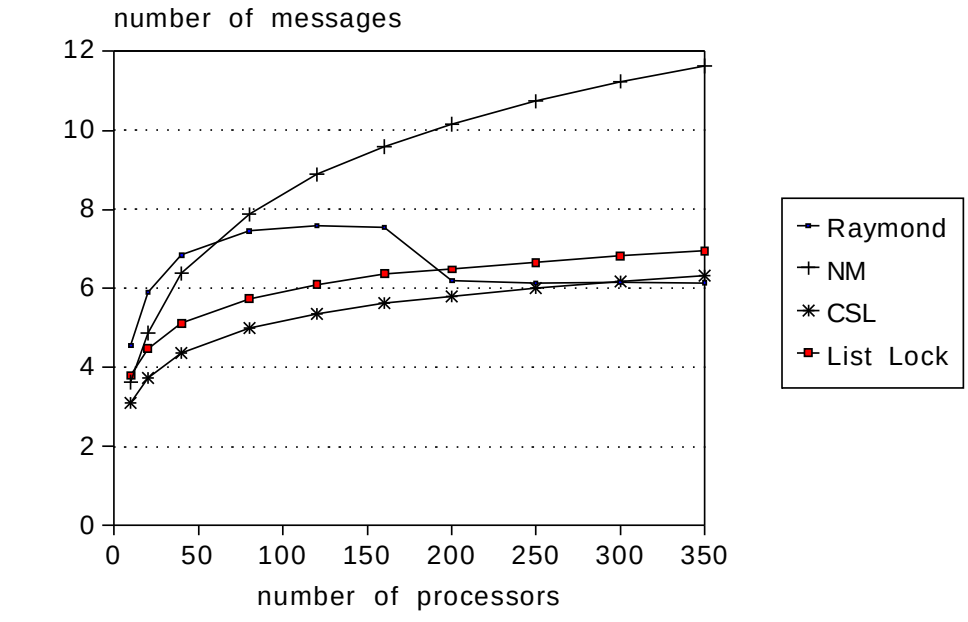

Figure 10: Number of messages per critical section entry, 50% load.

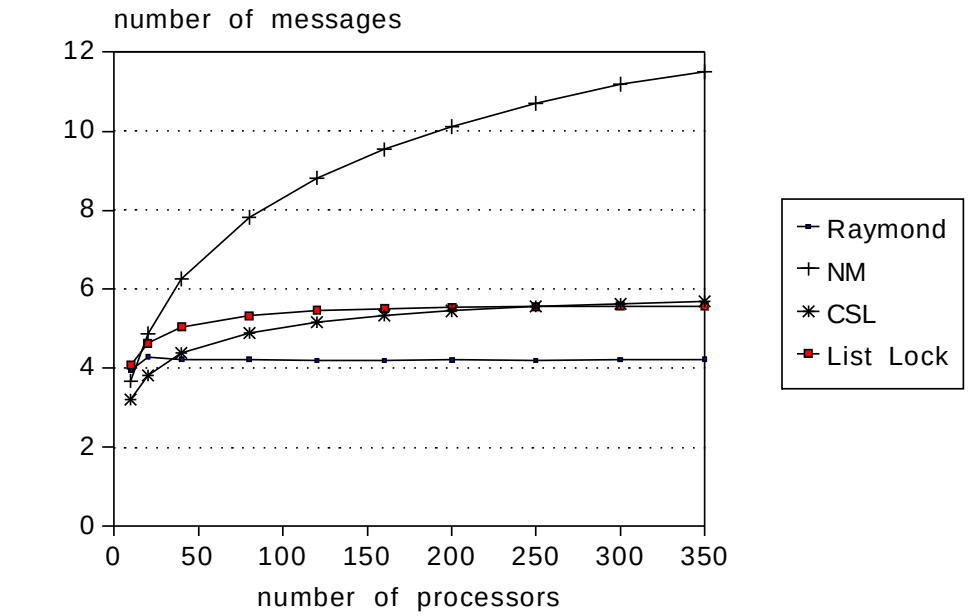

Figure 11: Number of messages per critical section entry, 50% load.

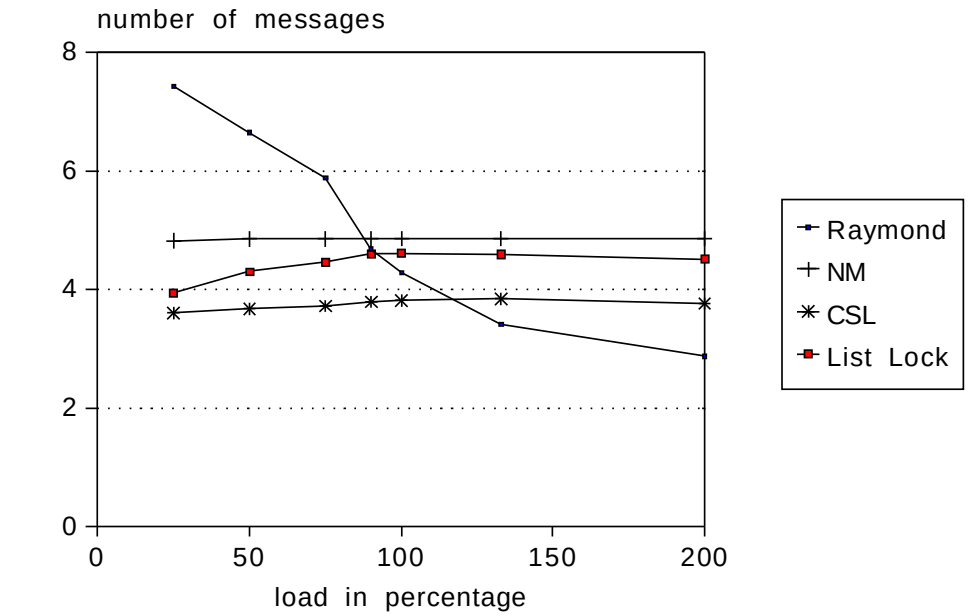

Figure 12: Number of messages per critical section entry, 20 processors.

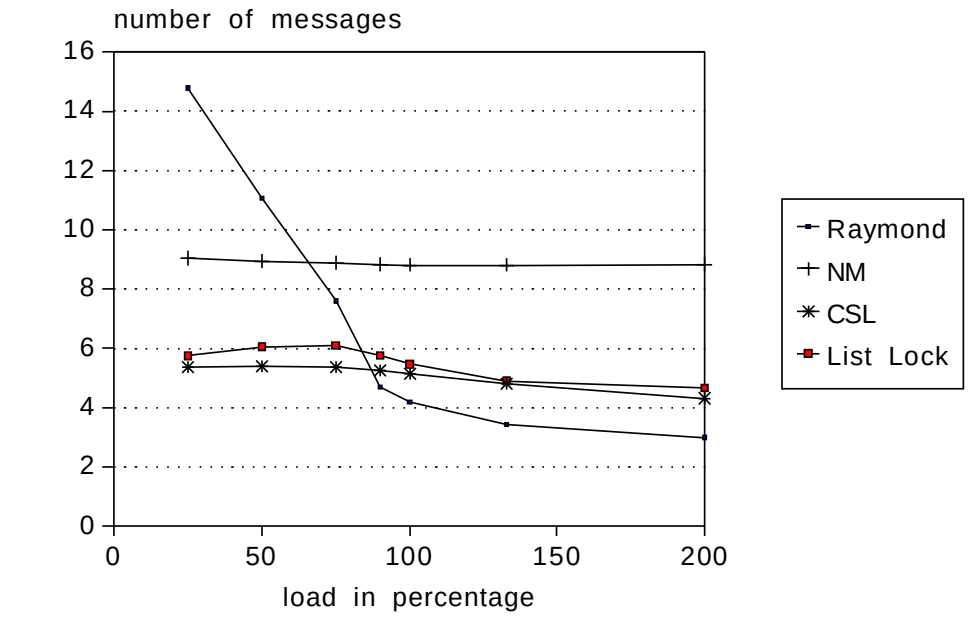

Figure 13: Number of messages per critical section entry, 120 processors.

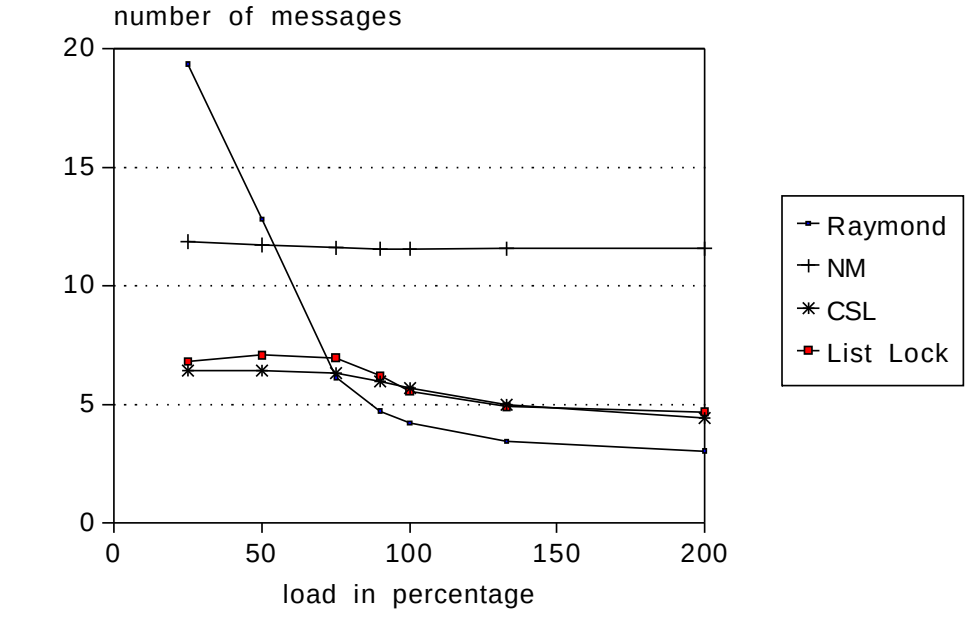

Figure 14: Number of messages per critical section entry, 350 processors.

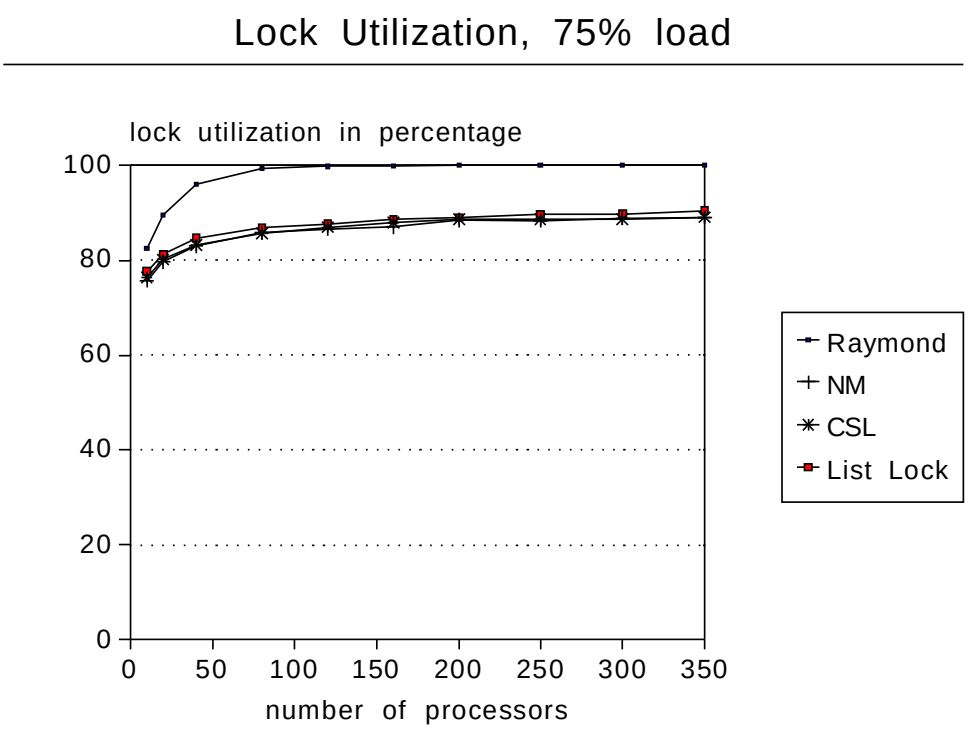

Figure 15: Lock utilization, 75% load.

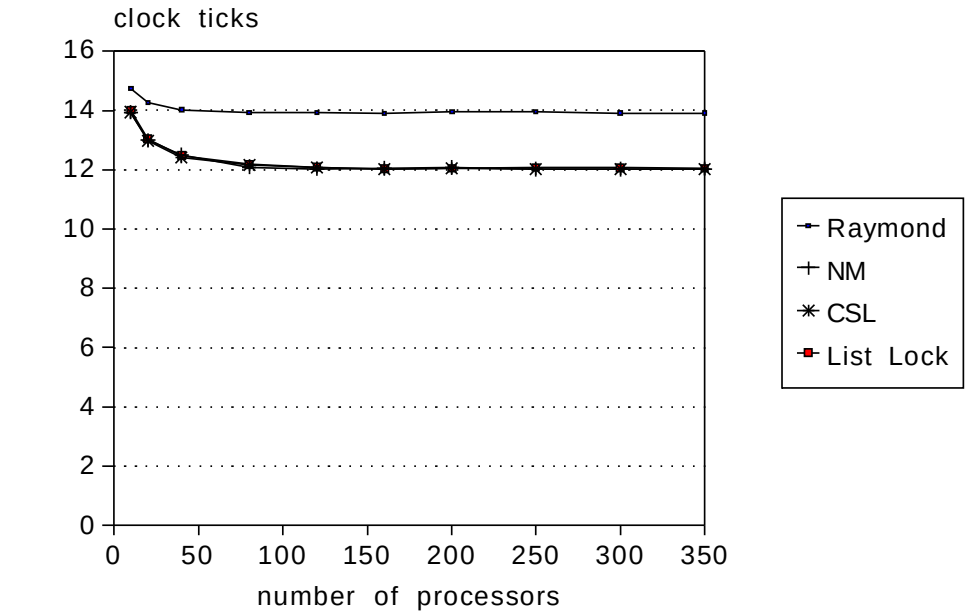

Figure 16: Critical section execution time, 100% load.

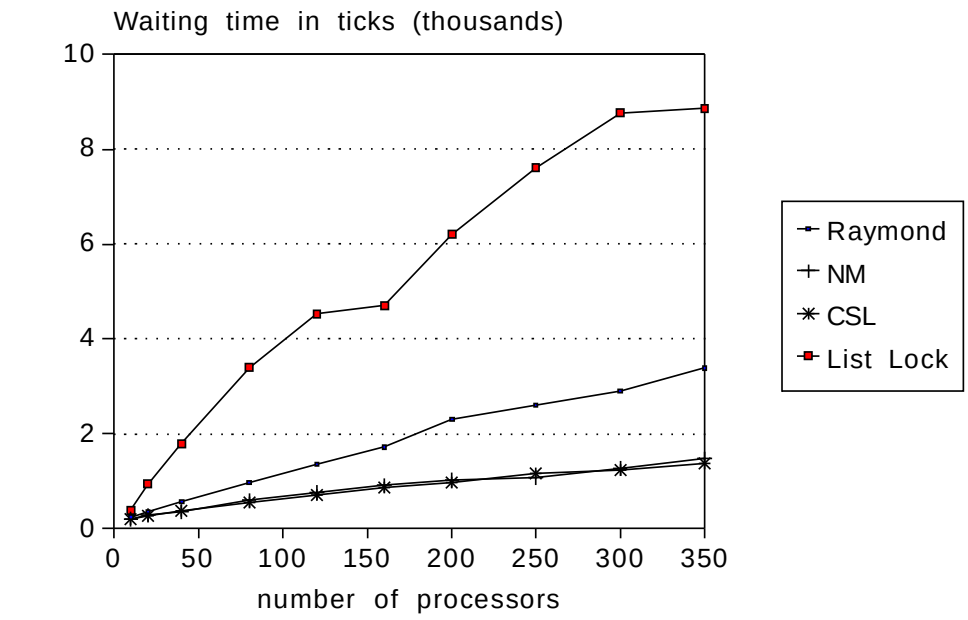

Figure 17: Maximum waiting time, 100% load.