Journal of Parallel and Distributed Computing **60**, 785–806 (2000)
doi:10.1006/jpdc.2000.1635, available online at http://www.idealibrary.com on IDEAL®

Empirical Evaluation of Mutual Exclusion Algorithms for Distributed Systems[1]

Shiwa S. Fu

Institute for Advanced Commerce, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598
E-mail: ssf@watson.ibm.com

Nian-Feng Tzeng

Center for Advanced Computer Studies, University of Louisiana at Lafayette, Lafayette, Louisiana 70504
E-mail: tzeng@cacs.louisiana.edu

and

Jen-Yao Chung

Institute for Advanced Commerce, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598
E-mail: jychung@watson.ibm.com

Received September 30, 1997; revised October 30, 1998; accepted January 28, 2000

Mutual exclusion in distributed memory systems is realized by passing messages among sites to establish a sequence for the waiting sites to enter the critical section. We have evaluated various distributed mutual exclusion algorithms on the IBM SP2 machine and the Intel *iPSC*/860 system, with their empirical results compared in terms of such criteria as the number of message exchanges and response time. The results take into account the effects of critical section request rate, critical section duration, and system size. Our results indicate that the Star algorithm (1991, M. L. Neilsen and M. Mizuno, in "Proc. 11th Int. Conf. Distributed Computing Systems," pp. 354–360) achieves the shortest response time in most cases among all the algorithms on a small to medium-sized system, when sites request the critical section many times before involving any barrier synchronization. This is because (1) it requires the exchange of no more than three messages per critical section entry, and (2) contention can quickly be alleviated after several entries into

[1] A preliminary version of this work was presented at the 11th International Parallel Processing Symposium, April 1997. Research was partially supported by NSF Grants MIP-9201308 and CCR-9300075. Correspondence may be sent to Dr. Shiwa Fu.

the critical section, if no barrier synchronization is involved in the meantime. On the other hand, if every site enters the critical section only once before encountering a barrier, the improved Ring algorithm (1995, S. S. Fu and N.-F. Tzeng, "Efficient Token-Based Approach to Mutual Exclusion in Distributed Memory Systems," Tech. Rep. TR-95-8-1, CACS, Univ. Southwestern Louisiana, Lafayette) is found to outperform others under a heavy load; but the Star algorithm and the CSL algorithm (1990, Y. I. Chang, M. Singhal, and M. T. Liu, in "Proc. 1990 Int. Conf. Parallel Processing," Vol. III, pp. 295–302) prevail when the request rate becomes light. The best solution to mutual exclusion in distributed memory systems is determined by how participating sites generate their mutual exclusion requests.

1. INTRODUCTION

Sites in a distributed system communicate by message exchanges through communication channels and do not share global memory. *Mutual exclusion* in a distributed system is achieved by a mechanism that ensures involved sites get access to a designated section of code (called the *critical section*) in a mutually exclusive way. Mutual exclusion has been widely applied to solve many problems, such as replicated data consistency [1, 2] and distributed shared memory [3]. In particular, the release consistency model [4] in distributed shared-memory systems heavily utilizes the concept of mutual exclusion.

To measure traffic overhead caused by message exchanges due to mutual exclusion, it is common to adopt the parameter of mean number of messages exchanged per critical section entry, or NME for short. An algorithm that leads to a smaller NME is preferable because it tends to yield lower traffic overhead. From the user's point of view, however, response time appears more important than NME. To allow its wide applicability, a mutual exclusion algorithm should scale well. In addition, an efficient such algorithm ought to avoid experiencing an exceedingly long response time when encountering various critical section request rates or critical section durations. The focus of this paper is on evaluating various distributed mutual exclusion algorithms on two real machines, comparing their behaviors in terms of NME and response time, and taking into account the effects of critical section request rate, critical section duration, and system size. Four algorithms are compared, including Raymond's algorithm [7], Neilsen and Mizuno's algorithm [8] with star topology (called the Star algorithm), the improved Ring algorithm [10], and Chang, Singhal, and Liu's algorithm (i.e., CSL in short) [9]. Our improved Ring algorithm is a variation of that described earlier in [24] but exhibits improved performance due to the elimination of unnecessary messages, as detailed in Section 2.4. In addition, we also introduce and evaluate a modified Raymond algorithm, which gives rise to better performance than the Raymond algorithm.

The behavior of distributed mutual exclusion algorithms is very complex and hard to analyze mathematically. It is difficult to obtain practical insights through theoretical analysis only. Most previous works thus limit their analytic scopes to parameters that can be analyzed, mainly NME, while employing simulation techniques to measure parameters that cannot be analyzed, such as response time. In general, analytic evaluation can predict the complexity of algorithms in terms of orders of magnitude; but it cannot clearly distinguish which algorithm outperforms others when algorithms (such as these investigated in this paper) all have the same order of NME complexity. As a result, early performance studies essentially relied on simulation for evaluating distributed mutual exclusion algorithms. Performance comparison through simulation, while remedying the shortcoming of theoretical analysis, has limitations on assumptions of traffic patterns and program execution, usually failing to reflect actual behaviors of algorithms and leading to impractical results. As a result, it is highly desirable to conduct an empirical study that actually implements these algorithms on real systems for performance comparison to overcome the shortcomings and limitations associated with simulation or analysis.

In order to observe the actual behaviors of these algorithms, we implemented them on the IBM's Scalable POWERparallel System 2 (SP2) [5] and the Intel *iPSC*/860, collecting empirical results under different critical section request rates and critical section durations. We carried out our study on the IBM SP2 machine of size up to 64 and on the Intel *iPSC*/860 of size 16 (which is the largest subcube available to users). Our results on both distributed-memory machines suggest that Neilsen and Mizuno's Star algorithm [8] outperforms all other algorithms with respect to response time for most cases, when the critical section is requested by sites repeatedly and no barrier synchronization is involved in the meantime. This is due to the following two facts: (1) the Star algorithm has the lowest NME unless the request rate is extremely high, and (2) while all sites contend for the critical section initially (since they start issuing their first critical section requests within a short interval), fewer and fewer sites experience contention gradually, as only one site is permitted to enter then leave the critical section at a time in sequence and a site does not generate another critical section request until its earlier request has been served. Consequently, sites gradually serialize their generation of critical section requests and soon avoid most contention, as long as no barrier synchronization is involved in the meantime. If every site issues just one request to the critical section before being involved in barrier synchronization, our improved Ring algorithm exhibits the best performance for a high request rate; but the Star and the CSL algorithms are superior to others for a low request rate.

Our experimental results shed some light on the design of distributed systems comprising computers interconnected by networks. In such a common distributed system configuration, critical section execution typically involves some operations that read from, or write to, remote memory locations. If the network for interconnection in the future is made to reduce latency, the critical section duration in such a distributed system becomes shorter. This will make the Star and CSL algorithms more attractive than others, as will be shown by our experimental results. The rest of this paper is organized as follows. Section 2 describes these algorithms considered in this study. In Section 3, the empirical results under different numbers of critical section entries are presented, whereas the performance outcomes under different numbers of critical section entries are discussed in Section 4. Pertinent work is described in Section 5. Conclusions are given in Section 6.

2. DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

Distributed mutual exclusion algorithms can be divided into two classes: (1) *permission-based* algorithms, where all involved sites vote to determine which site receives the permission to enter the critical section, and (2) *token-based* algorithms, in which only the site with the token may enter the critical section. For an algorithm in class (1), a site must obtain the permission from a quorum of involved sites in the same subset before entering the critical section. A system may consist of only one set, which contains all sites [11, 12–14, 35], or multiple subsets [15–17]. A class (2) algorithm, on the other hand, manages a unique token and guarantees the site acquiring the token to enter the critical section. There are many algorithms in this class [7–10, 18–22, 24, 34, 37].

For permission-based algorithms, each site typically communicates with most of the sites and keeps the status information of many sites, creating excessive message exchanges and involving high storage overhead. In general, a permission-based algorithm involves higher communication traffic overhead than a token-based algorithm, because the latter often communicates with fewer sites before entering the critical section, as validated by Chang's simulation results [36]. We therefore focus our attention to token-based algorithms in our empirical study. Among known token-based algorithms, five of them have been shown to achieve better performance, with their average NME values all being $O(\log N)$ per critical section entry [7–9, 19, 37]. The simulation study in [36] compared a variety of permission-based algorithms [12, 14, 15] and token-based algorithms [7, 19–21, 37], indicating that the algorithm by Naimi and Trehel [19] exhibits better NME and time delay. This algorithm [19] adopted the token queue concept, where a queue contains the identifiers of the sites requesting the critical section and the queue is attached to the token message when it travels around the system. The disadvantages of this algorithm are that (1) the size of the token queue is not fixed and could grow up to $N-1$ in a system with N sites, and (2) time overhead to prepare and process the token message and maintain local information at each site is high. This token queue overhead has been overcome by the CSL algorithm [9]. To avoid duplicated work with the previous studies in [9, 36], we therefore selected the CSL algorithm [9] in our study among those described by [9, 19, 37]. Additionally, in considering a variety of measurement issues, three other algorithms [7, 8, 10] are chosen, as stated in sequence. Both the Raymond and the CSL algorithms have the same NME complexity of $O(\log N)$ but the former employs a static structure while the latter assumes a dynamic underlined structure. To compare and contrast the impact of logical structures on performance, we included Raymond's algorithm [7] in our portfolio. To measure the impacts of system size, request rate, and critical section duration on performance of algorithms with different orders of magnitude in complexity, we selected the improved Ring algorithm [10]. Although Banerjee's algorithm in [34] has the same NME complexity of $O(N)$ as the improved Ring algorithm does, the latter has a wider range of NME values, namely 2 in the best case and $2(N-1)$ in the worst case (in contrast to 3 and N, respectively, in Banerjee's algorithm), making it an ideal candidate to illustrate issues mentioned above. We also employed Neilsen and Mizuno's algorithm [8] in our study, since their fixed star topology has the merits of a simple structure and easy implementation, potential to exhibit better performance than other algorithms in most cases. The algorithms under this study are reviewed briefly below.

2.1. Raymond's Algorithm

Raymond's algorithm [7] determines and maintains a *static* logical structure. The logical structure (for example, a spanning tree) is kept unchanged throughout its lifetime, but the directions of edges in the structure change dynamically as the token migrates among sites, in order to point toward the possible token holder. The token moves in response to a token request issued by a site wishing to enter the critical section, and it stays at its current site until a token request arrives. When the token travels over a link (in response to a token request) toward the requesting site, the direction of the link is reversed because the requesting site will eventually become the new token holder after receiving the token. As a result, the directions of edges in the structure always point to the possible token holder, making the token holder a sink node in the structure. Each site has a local queue to hold requests coming from its neighbors and itself. When a request message arrives at a site that has already issued a request, no further message is sent by the site, as the token message will be drawn to the site (by the earlier request). Under such a circumstance, each site has only one outstanding request at any given time, resulting in the local queue length no more than the node degree of the embedding structure. This property eliminates unnecessary exchange messages and keeps the amount of communication traffic low under any request load.

Each site wishing to enter the critical section inserts its local request to the rear of its local queue, so that all requests that appear at that site are in a first-come-first-served order. While it is possible to obtain better performance by inserting a locally generated request at the front of the local queue, referred to as the eager Raymond algorithm (because the local site is then allowed to enter the critical section immediately when the token reaches the site) [7], this tends to pose a concern on the fairness of requests being served and is thus not considered here.

2.2. Modified Raymond Algorithm

On seeing the token, an intermediate site (which is not waiting for the critical section) in Raymond's algorithm determines the link along which the token is to be forwarded, by examining the head request in its local queue. The head request is then dequeued, once the token is forwarded along. If the local queue is not empty after dequeueing (indicating that some site(s) in the other branch is (are) waiting for the token), a token request is produced and sent along the link taken by the token, in order to get the token back after all waiting sites in the branch get served [7]. In other words, a token request always follows the token from an intermediate site whose local queue contains more than one element. This situation happens more frequently as the critical section request rate grows.

We introduce a simple modification, as shown in Fig. 1, to lower communication traffic by eliminating the token request from a site whose local queue (i.e., Req_Q) contains multiple elements. Instead of sending a separate token request (i.e., REQT

```
enqueue(Req_Q, Me);
if (HOLDER == Me) {
    HOLDER = dequeue(Req_Q);
    if (HOLDER == Me) {
        enter critical section;
    }
    else
        send TOKN msg is marked to HOLDER;
}
else if (no REQT msg or marked TOKN msg has been issued)
    send a REQT msg to HOLDER;
```

(a) Request critical section.

```
enqueue(Req_Q, Y);
if (HOLDER == Me) {
    if (not in critical section) {
        HOLDER = dequeue(Req_Q);
        if (Req_Q is not empty)
            send TOKN msg is marked to HOLDER;
        else
            send a TOKN msg to HOLDER;
    }
}
else if (no REQT msg or marked TOKN msg has been issued)
    send a REQT msg to HOLDER;
```

(b) Receive a REQT msg from neighbor Y.

```
if (Req_Q is not empty) {
    HOLDER = dequeue(Req_Q);
    if (Req_Q is not empty)
        send TOKN msg is marked to HOLDER;
    else
        send a TOKN msg to HOLDER;
}
```

(c) Exit critical section.

```
HOLDER = Me;
if (TOKN msg is marked) enqueue(Req_Q, Y);
if (Req_Q is not empty) {
    HOLDER = dequeue(Req_Q);
    if (HOLDER == Me) {
        enter critical section;
    }
    else {
        if (Req_Q is not empty)
            send TOKN msg is marked to HOLDER;
        else
            send a TOKN msg to HOLDER;
    }
}
```

(d) Receive a (marked) TOKN msg from neighbor Y.

FIG. 1. Modified Raymond algorithm.

message), the site marks in the token message the situation that the token has to come back later on (see the 4th last lines in Figs. 1a and 1c, the 7th last line in 1b and the 5th last line in 1d). A marked token causes an enqueueing operation at the receiving site (the 2nd line in 1d), recording that the token will be sent back along the link from which it gets to the site. This combines the token message with a subsequent token request message at every site whose local queue length is greater than 1, effectively lowering mutual exclusion traffic and thus improving performance.

2.3. Neilsen and Mizuno's Star Algorithm

Instead of passing the token step by step through intermediate sites in the logical structure to the token requestor as in Raymond's algorithm [7], Neilsen and Mizuno proposed an algorithm where the token holder can send the token directly to the requesting site with one message [8]. This is made possible by attaching the requestor's ID in the request message so that the token holder knows, on receiving the message, who is the requestor.

One special case of this algorithm is that the logical structure can be a fixed star topology (called the Star algorithm). Under such a situation, the root site makes it possible to establish a distributed waiting queue (of all requesting sites) by recording the site which has most recently requested the token (and is the tail site in the distributed waiting queue). Any site (except the root) ready to enter the critical section always sends a request message attached with its own ID directly to the root site (of the star topology), whereas the root site sends its message to the tail site of the queue. When receiving a request message piggybacked with a site ID, the root forwards the message to the tail site (of the queue) and updates its record, unless the root itself holds the token. A token holder, after exiting from the critical section, forwards the privilege to the next waiting site directly using a token message. The token hence traverses along the requesting sites in a sequence dictated by the waiting queue. When the queue is empty, the token stays at the site which entered the critical section latest, until the site receives a token request message. A very attractive property of the Star algorithm is that it always takes three exchange messages for a requestor to get the token, if the root does not own the token, and only two messages if the root holds the token. In this algorithm, the concept of forming a distributed waiting queue of all requesting sites follows a similar idea as in [23] to solve mutual exclusion in shared-memory multiprocessor systems. Mutual exclusion in shared memory systems is achieved by accessing the global shared memory instead of passing messages as in the distributed-memory systems. Evaluation of software-based mutual exclusion algorithms for shared memory systems was performed in [31], with the effects of architectures and systems taken into account.

2.4. *Improved Ring Algorithm*

The algorithm proposed in [24] establishes a static logical ring over all sites and allows the token to move along a fixed direction, in response to a token request, which travels along an opposite ring direction. The logical ring and the directions of its links are all kept unchanged, making the algorithm simple and easy to implement. When ready to enter the critical section, a site without the token, say S_w, must request the token from the current token holder using a request message; site S_w has the privilege to enter the critical section as soon as it receives the token. After issuing a request message, S_w goes to the WAIT (short for *waiting*) state until it receives the token. Since S_w has no knowledge about the current token holder, it always sends the request message to its successor, site $S_{(w+1) \bmod N}$, which then becomes the *substitute* site of S_w (for requesting the token) after receiving the request message, where N is the system size. If $S_{(w+1) \bmod N}$ is not the token holder, it sends a request message to its successor, site $S_{(w+2) \bmod N}$, which again becomes the substitute site of S_w (for requesting the token). This process repeats until the site with the token, say S_h, receives a request message from its predecessor and the sites within S_w and S_h (along the direction of the request message traversals) are all at the SUBS (short for substitute) state. On receiving a request message, the token holder, if not in need of the token, forwards the privilege to its predecessor using a token message. The token is then forwarded by the SUBS sites in sequence to site S_w (along the reverse direction of the request message). If the number of SUBS sites is α, $0 \leqslant \alpha < N-1$, the total number of messages exchanged for S_w to obtain the privilege of entering the critical section equals $2(\alpha+1)$.

We have introduced a variation of the Ring algorithm to achieve improved performance [10]. The pseudocode of this algorithm is illustrated in Fig. 2, where each site can be in one of the four possible states: IDLE, SUBS, WAIT, and EXCU (executing the critical section). A site is in the IDLE state initially, and it goes to the SUBS state after receiving a REQT message, if it is not the token holder. When a site wishes to enter the critical section, it moves to the WAIT state, if it is not the token holder; otherwise, it goes to the EXCU state, starting to execute the critical section. The improvement of this algorithm results from the following two facts: (1) it is possible that SUBS sites may also wish to enter the critical section in the

```
if (holding token) {
    state = EXCU;
    enter critical section;
}
else {
    if (state == IDLE)
        send a REQT msg to successor;
    state = WAIT;
}
```

(a) Request critical section.

```
if (holding token) {
    if (state != EXCU)
        send TOKN msg to predecessor;
    else SUBS_FLAG = true;
}
else {
    SUBS_FLAG = true;
    if (state == IDLE) {
        state = SUBS;
        send a REQT msg to successor;
    }
}
```

(b) Receive a REQT msg from predecessor.

```
state = IDLE;
if (SUBS_FLAG == true) {
    SUBS_FLAG = false;
    send TOKN msg to predecessor;
}
```

(c) Exit critical section.

```
if (state == WAIT) {
    state = EXCU;
    enter critical section;
}
else {                    /* state == SUBS */
    state = IDLE;
    SUBS_FLAG = false;
    send TOKN msg to predecessor;
}
```

(d) Receive TOKN msg from successor.

FIG. 2. Improved Ring algorithm.

meantime, and such a site is allowed to do so by holding the token until it finishes with the critical section after it receives the token message from its successor, and (2) a SUBS site does not have to issue any message to its successor for requesting the token, because the token will traverse the site on its way to S_w (i.e., the token requestor), and the site can enter the critical section when it observes the token. A SUBS site ready to enter the critical section goes to the WAIT state, and a site at the WAIT state is allowed to enter the critical section on receiving the token [10]. In the best scenario, as many as $(\alpha+1)$ sites (i.e., those α SUBS sites and S_w) may enter the critical section in sequence with $2(\alpha+1)$ messages exchanged, giving rise to NME $=2$. This attractive property occurs under a heavy request load for the critical section. However, the NME could be as large as $2(N-1)$ for a system with N sites in the worst case, as the token message might travel one revolution (in response to the request message) to serve a requestor under a very light request load. Since the token message moves along a fixed direction, each token requestor can be served within one revolution; consequently, the potential last-come-first-served problem in the sites transiting from the SUBS state to the WAIT state does not pose much concern in this algorithm. Our experimental study considered this variant ring algorithm, referred to as the improved Ring algorithm [10], for it results in better performance than that proposed in [24].

2.5. Chang, Singhal, and Liu's Algorithm

The algorithm by Naimi and Trehel [19] employed a *dynamic* logical structure, which keeps changing during execution. It requires a local queue to keep the identifiers of all sites ready to enter the critical section, and the token message carries the queue when it travels around the system. As a result, the size of the token message is not fixed and could grow up to $N-1$ in a system with N sites. This is an undesirable property. Chang, Singhal, and Liu considered an algorithm [9] to overcome this drawback by maintaining a list that links all requesting sites (i.e., a distributed queue) such that each requesting site records (using variable **Next**) only the identifier of its next requesting site, thereby simplifying the data structure of the token message [9]. The logical structure in the CSL algorithm is a star topology initially, and it changes dynamically as the algorithm proceeds. A site is the tail in the distributed queue, if it is waiting for the token and its **Next** is **NIL**. If its **Next** is not **NIL**, its successor site in the distributed queue is pointed by **Next**. As a result, when a token request message arrives at a site which is the tail in the distributed queue, the site simply sets its **Next** to the requesting site. If a request message arrives at a site which neither holds nor is requesting the token, or which is requesting the token but its **Next** is not **NIL**, the request message is forwarded to the possible token holder (pointed by variable **NewRoot**) to form a distributed queue; **NewRoot** is then set to point to the current requestor because it will eventually becomes the new token holder. On sending the token message to the next site, the variable **NewRoot** is piggybacked in the token message so that the next site can update its **NewRoot** accordingly.

The NME complexity of this algorithm depends on the height of the logical tree, and it is $O(\log N)$ per critical section entry, where N is the system size.

2.6. Summary

A summary of the four algorithms considered in this study is provided in Table 1, where the NME complexity of Raymond's algorithm is $O(\log N)$ and so are the modified Raymond algorithm and the CSL algorithm; it is $O(N)$ for the improved Ring algorithm, and is no more than 3 for the Star algorithm. The token message and the request message used to communicate among sites are of fixed sizes for all these algorithms. In our implementation, both types of messages by default contain four words (i.e., the source identifier, the destination identifier, the message type, and the communicator which specifies the set of processes that share this communication context). They are five words in the CSL algorithm due to carrying one extra word of identifiers. The request message in the Star algorithm contains five words owing to carrying one extra word of identifiers.

TABLE 1

Comparison of Various Algorithms

	Raymond [7] Modified Raymond	Improved Ring [10]	CSL [9]	Star [8]
NME	$O(\log N)$	$O(N)$	$O(\log N)$	$\leqslant 3$
Token message size	Fixed	Fixed	Fixed	Fixed
Request message size	Fixed	Fixed	Fixed	Fixed
Logical structure	Static	Static	Dynamic	Static
Local queue at each site	$\leqslant$ Node degree of logical structure	No	No	No
Implementation difficulty	Difficult	Easy	Moderate	Easy
Load balance	Worse	Good	Good	Worse

Only the CSL algorithm maintains a *dynamic* logical structure, while others construct *static* logical structures throughout their lifetimes. The Raymond algorithm and the modified Raymond algorithm require a local queue in each site to hold the requests (with totally no more than the node degree of the logical structure), but all three other algorithms require no local queue at any site. From the implementation standpoint, Raymond's algorithm is the most difficult to implement, followed by the CSL algorithm, as the former must maintain the local queue and adjust the directions of edges in the logical structure when the token moves, while the latter constantly keeps the dynamic structure. Both the improved Ring and the Star algorithms have a simple and fixed logical structure which is easy to implement. When the load balance in each node is taken into account, both the Star and the Raymond algorithms have worse load balance with the nodes closer to the center of the logical structure having higher NME values. However, all nodes in the Star algorithm, according to our empirical results, have the same response time under different request loads and system sizes, indicating that the load imbalance at the center node does not degrade its performance. The CSL and the improved Ring algorithms have good load balance across nodes due to their respective dynamic and uniform logical structures. The performance evaluation of these algorithms is provided in the next section.

3. PERFORMANCE EVALUATION

Our experiment was conducted on both the IBM SP2 machine [5] at Argonne National Laboratory (ANL) and the Intel *iPSC*/860 system at Mississippi State University (MSU).

3.1. Platforms

The SP2 involves 128 nodes, which are standard POWER2 Architecture RISC System/6000 processors with speed of 66.7 MHz and are interconnected by a multistage network similar to the Omega network. Each node has its own local memory, and communication among nodes is achieved by message passing. In our study, the message-passing interface (MPI) [25] standard supported by SP2 is utilized to achieve communication between any two nodes, and the nonblocking point-to-point MPI functions are used. The latency for sending a message from one node to another involves (1) software overhead to initiate a send operation, and (2) hardware overhead of $46 + 0.035m\,\mu$s [26], where m is the message size in bytes. According to our measurement on the IBM SP2 at ANL, it takes about 110 μs to send a 1-byte message to another node in the system.

The Intel *iPSC*/860 is a distributed memory system, with its nodes interconnected according to the hypercube topology. Each cube node is a 40-MHz *i*860 processor with 8 Mbytes of memory. Communication and coordination among nodes are through message-passing. Again, the time taken to deliver a message includes software overhead and hardware overhead, which can be expressed by $95 + 0.394m + 10.3d\,\mu$s for m bytes over distance d [29], with the distance between two neighboring nodes being 1. Our measurement on the *iPSC*/860 reveals that it takes roughly 150 μs to transfer a 1-byte message between two nodes of 1 unit distance. The communication time on the *iPSC*/860 is larger than that on the IBM SP2. While the *iPSC*/860 machine at MSU contains 32 nodes, a user may request up to only 16 nodes (a 4-dimensional subcube) at a time.

3.2. Implementation Details

The static logical structure (i.e., a ring) for the improved Ring algorithm is embedded in a straightforward way in which site S_x is connected to site $S_{(x+1) \bmod N}$, where x is the logical ID of a particular node and N is the allocated system size. The token is assigned to an arbitrary site initially. For the (modified) Raymond algorithm, a binary spanning tree is embedded across involved nodes according to Prim's method stated in [27] with some modifications, and an arbitrary node is selected as the root. A star topology is embedded for both the CSL algorithm and the Star algorithm after the token is assigned to an arbitrary site as the root. Note that the logical structure (a star initially) in the CSL algorithm changes dynamically as the algorithm proceeds, while it remains unchanged for the Star algorithm. The Star algorithm in our implementation follows that described in [8] with a slight modification to maintain the fixed star structure throughout its lifetime.

Workload and performance metrics. In our experiment, every involved site produces mutual exclusion requests in a random fashion governed by a random number generator with a mean value of Γ (which is the mean time between releasing the critical section and requesting it again by a typical site). Note that this way of generating mutual exclusion requests corresponds to many real situations where a site cannot proceed until its issued mutual exclusion request is served. Consequently, the degree of contention reduces gradually, as long as there is no barrier synchronization involved, since sites enter and leave the critical section one by one. The time taken by a site to execute the critical section is specified by ξ. Our experimental results were collected under a wide range of workloads with Γ chosen from 300 to 18,000 μs and ξ from 100 to 1000 μs. A larger Γ value reflects a lighter request load, while a small ξ mirrors the situation that the critical section contains a few instructions.

The performance measures of interest include (1) NME and (2) the response time, which is defined as the average time from requesting the token to finishing the critical section by a site. A small NME value indicates light traffic overhead, while a short response time reflects assigning the critical section to waiting sites effectively. In evaluating these two performance measures, we take into account the effects of critical section request interval (Γ), critical section duration (ξ), and system size, as demonstrated in the following subsection.

3.3. Empirical Results and Discussion

We first examine the impacts of the critical section request interval (Γ) and the critical section duration (ξ) on system performance. All sites execute the same code and stop execution after any one of them reaches 10,000 critical section entries (by broadcasting a stop message to terminate the remaining sites). The results illustrated in Figs. 3–8 are averaged values over all the involved sites.

A program code typically consists of the critical section part and the noncritical section part. Both parts are likely to have some read/write operations, and the critical section normally contains a few instructions. In a distributed memory system (like SP2 or *iPSC*/860), a read from or write to a remote memory location takes at least one message sending time, which is about 110 μs on SP2 and about 150 μs on *iPSC*/860. For convenience, the critical section duration (ξ) in our experiment is set to multiples of 100 μs.

Effect of critical section request interval. In Fig. 3, NME and response time versus Γ obtained on SP2 under system size $N = 16$ are illustrated for these algorithms, where ξ is chosen as 100 μs and Γ ranges from 300 to 18,000 μs. The improved Ring algorithm has the lowest NME value (of 2) under a heavy request rate (i.e., $\Gamma \leqslant 1000$ μs), but its NME grows quickly as Γ increases. This is because most intermediate sites are in the WAIT state under a heavy request rate, but fewer sites are in the WAIT state as the request rate decreases. The NME values of Raymond's and the modified Raymond algorithms grow as Γ increases, but they flatten off after $\Gamma > 6000$ μs (or 12,000 μs). The modified version has a smaller NME than the original Raymond algorithm for any Γ, as expected. When compared with the improved Ring algorithm under a heavy request rate (say $\Gamma \leqslant 3000$ μs), the modified Raymond algorithm yields worse performance on both NME and response time. The reason is that each site in the improved Ring algorithm then requires as few as two messages (one request message and one token message) to accomplish one mutual exclusion entry, whereas each site in the tree structure of the modified Raymond algorithm requires more messages to do so, because the token message in the tree structure travels (in response to a request message) along each of the $(N-1)$ edges exactly twice in order to bring the token to all N sites in the tree.

It is observed from Fig. 3 that NME for the CSL algorithm increases slightly at the beginning but remains flat afterward (i.e., $\Gamma > 3000$ μs). Although both the Raymond and the CSL algorithms have the same NME complexity of $O(\log N)$, the CSL algorithm yields better performance in regard to NME than the modified

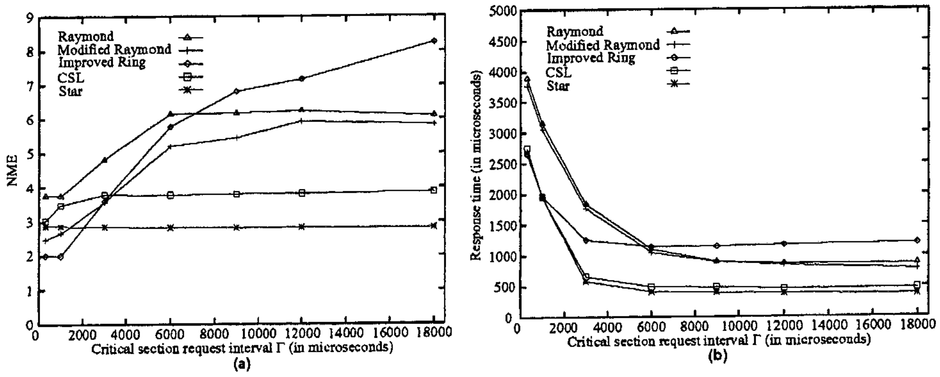

FIG. 3. NME and response time versus Γ on the IBM SP2 with $N = 16$ and $\xi = 100$ μs.

Raymond algorithm under a moderate to light request load, indicating that the dynamic logical structure used in the CSL algorithm can effectively adjust and evolve according to the occurrence of critical section requests, while the token message in the static structure used in the modified Raymond algorithm must move step by step through intermediate sites, which may not request the critical section in the meantime. However, the situation reverses under a heavy request load because most of the intermediate sites are then ready for the critical section, while sites in the CSL algorithm still need to forward request messages to form a distributed queue. The Star algorithm keeps a fixed NME value (of about 3) throughout the simulated Γ values, as expected.

The response time is dictated by Γ as shown in Fig. 3b. Under a heavy request rate (i.e., a small Γ value), many sites wait to enter the critical section and a long waiting queue results, thus prolonging response time. The response time curves of all these algorithms decrease gradually before flattening off as Γ increases. The Star algorithm has the best performance in regard to response time under almost all the Γ values (except for $\Gamma = 300\,\mu$s). The curve of the CSL algorithm stays close to that of the Star algorithm throughout the range of Γ examined. The improved Ring algorithm leads to the shortest response time when $\Gamma = 300\,\mu$s (due to its low NME), but it has the longest response time as Γ goes beyond $6000\,\mu$s, since the token message must then go through many intermediate sites to reach the requestor. The gap between the Star algorithm (or the CSL algorithm) and the improved Ring algorithm (and the modified Raymond algorithm) tends to be large.

NME and response time versus Γ under these algorithms on *iPSC*/860 with $N = 16$ are shown in Fig. 4, where ξ and the Γ range are selected as above. Again, the improved Ring algorithm presents the smallest NME value for a high request rate, but its NME increases consistently and much faster than any other algorithm if Γ grows. As might be expected, our modified Raymond algorithm always yields better performance than the original Raymond algorithm. The improved Ring algorithm is superior to the modified Raymond algorithm (in terms of both NME and response time) under a heavy request rate (say $\Gamma < 2000\,\mu$s). The Star algorithm exhibits better performance than the CSL algorithm throughout the Γ range considered. In fact, the Star algorithm always leads to the shortest response time among all algorithms, despite the fact that it has a slightly larger NME value than

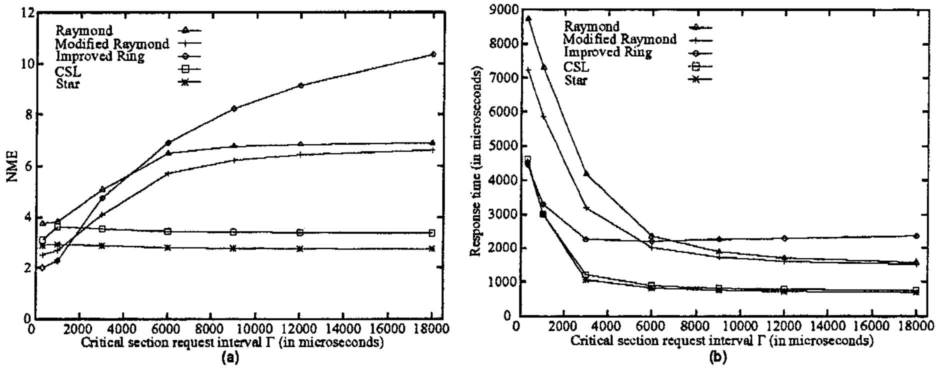

FIG. 4. NME and response time versus Γ on the Intel *iPSC*/860 with $N = 16$ and $\xi = 100\,\mu$s.

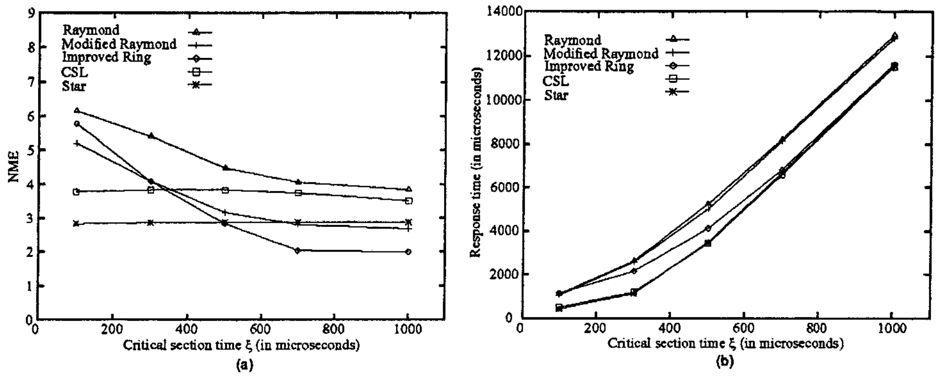

FIG. 5. NME and response time versus ξ on IBM SP2 with $N = 16$ and $\Gamma = 6000\,\mu$s.

the improved Ring algorithm or the modified Raymond algorithm when the request rate is very high. When comparing Figs. 3b and 4b, we find that the response time of a given case is more on *iPSC*/860 than on SP2, since communication takes longer on *iPSC*/860.

Effect of critical section duration. The effect of the critical section duration (ξ) on NME and the response time for SP2 is depicted in Fig. 5, where $N = 16$ and $\Gamma = 6000\,\mu$s. As can be seen, a larger ξ reduces the NME value but increases the response time for all algorithms. Especially, the NME for the improved Ring algorithm decreases dramatically when ξ increases, because more intermediate sites are then in the WAIT state for a larger ξ. In general, the improved Ring algorithm and the (modified) Raymond algorithm experience more pronounced reduction in NME than the other two algorithms when ξ grows, as can be seen in Fig. 5a. An increase in ξ causes more sites to wait for the critical section (i.e., a longer waiting queue), thus prolonging response time. Our modified Raymond algorithm outperforms Raymond's algorithm with respect to both NME and response time. The Star algorithm and the CSL algorithm yield shorter response times for all ξ values. It is interesting to observe that the response time for the improved Ring algorithm is close to that of the Star algorithm (and the CSL algorithm) after $\xi > 700\,\mu$s. This is mainly due to its extremely low NME value for a large ξ (see Fig. 5a). It is apparent from Fig. 5b that the Star algorithm and the CSL algorithm outperform other algorithms in terms of response time.

NME and response time as a function of the critical section duration (ξ) for *iPSC*/860 with $N = 16$ is shown in Fig. 6, where Γ equals $6000\,\mu$s. It is observed that an increase in ξ leads to a smaller NME value for the (modified) Raymond and the improved Ring algorithms, but a slightly larger NME value for the Star and the CSL algorithms. However, the response time grows monotonically as ξ increases for all the algorithms considered. While the Star algorithm maintains the smallest NME value until ξ grows to $920\,\mu$s but is inferior (in terms of NME) to the improved Ring algorithm afterward, it consistently leads to the shortest response time (which is very close to the response time of the CSL algorithm).

Results on SP2 with different sizes. To observe the results of these algorithms on a larger system, we conducted experiments on the IBM SP2 machine with

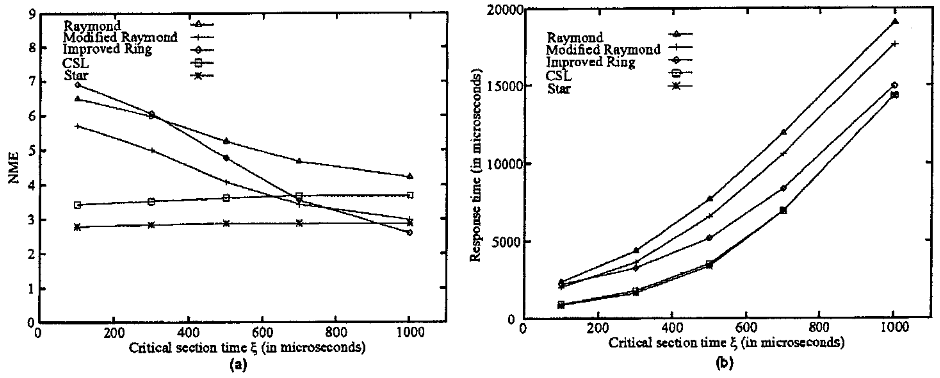

FIG. 6. NME and response time versus ξ on Intel *iPSC*/860 with $N = 16$ and $\Gamma = 6000\,\mu$s.

$N = 32$ (note that the maximum available subcube on our Intel *iPSC*/860 machine is of size 16 only). The collected NME and response time versus Γ are illustrated in Fig. 7, where ξ is 100 μs and Γ ranges from 300 to 18,000 μs. When compared with those for $N = 16$ shown in Fig. 3, the NME curves of the (modified) Raymond algorithm in Fig. 7 grow more quickly as Γ increases. The CSL algorithm leads to a larger NME value (of 4.5) in Fig. 7 than that in Fig. 3, when Γ grows to 18,000 μs. On the other hand, the Star algorithm has a fixed, small NME of roughly 3 for both $N = 16$ and 32, revealing its good scalability.

When compared to the response time curves for $N = 16$ shown in Fig. 3b, the curves depicted in Fig. 7b follow a similar trend but have larger values. Under a heavy request rate ($\Gamma \leqslant 1000\,\mu$s), for example, every algorithm under $N = 32$ experiences at least two times of response time of that under $N = 16$ (see Fig. 3), because the waiting queue then becomes longer when the system size (N) increases. The curves for all algorithms flatten off as Γ increases. In the case of $\Gamma = 9000\,\mu$s, the increases in response time (when compared with that under $N = 16$) is about 24% for the Star algorithm, 26% for the CSL algorithm, 49% for the improved Ring algorithm, and 65–67% for the (modified) Raymond algorithm, indicating that both the Star algorithm and the CSL algorithm exhibit better scalability.

To investigate the behaviors of these algorithms on various system sizes, we carried out our experiment on SP2 of size up to 64, with NME and response time

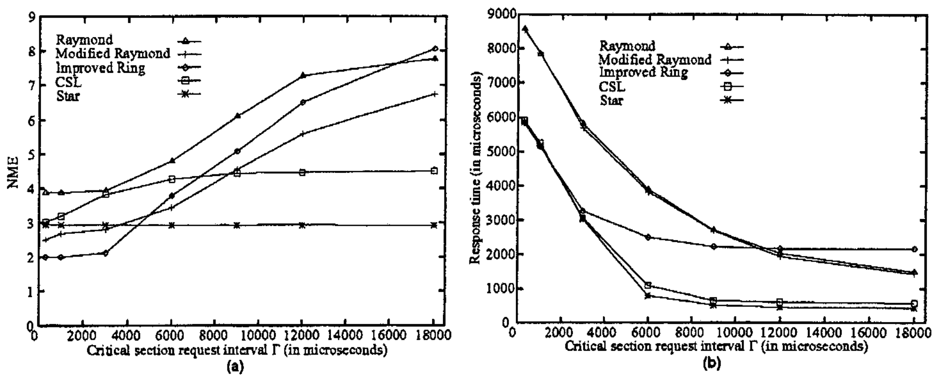

FIG. 7. NME and response time versus Γ on the IBM SP2 with $N = 32$ and $\xi = 100\,\mu$s.

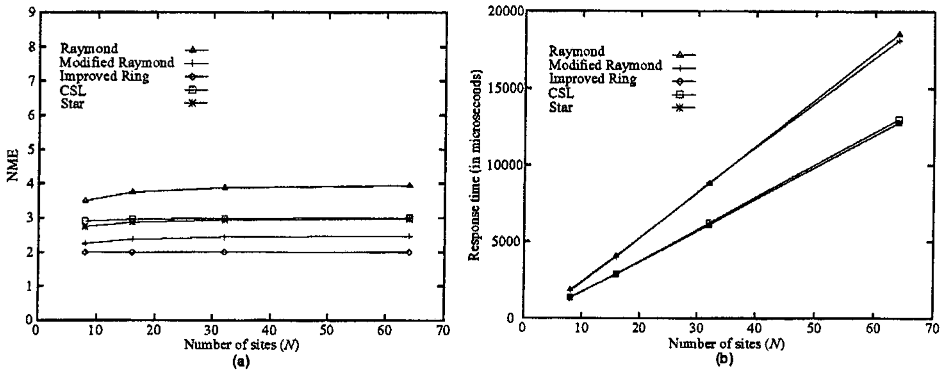

FIG. 8. NME and response time versus system size on SP2 under $\Gamma = 100\,\mu$s and $\xi = 100\,\mu$s.

results as a function of N illustrated in Fig. 8, where the mutual exclusion request interval (Γ) and the critical section duration (ξ) are both set to $100\,\mu$s. Each site requests the critical section very frequently under this Γ value, i.e., an extremely heavy request rate. Every algorithm presents an almost fixed NME throughout the considered range of N under this heavy request rate, as demonstrated in Fig. 8a. The improved Ring and the modified Raymond algorithms have the lowest NME values, whereas the CSL and the Star algorithms give rise to $\text{NME} = 3$. It is observed from Fig. 8b that the Star algorithm, the CSL algorithm, and the improved Ring algorithm exhibit shorter response times consistently, while both the Raymond and the modified Raymond algorithms have much longer response times.

4. PERFORMANCE UNDER DIFFERENT NUMBERS OF CRITICAL SECTION ENTRIES

The previous subsection displays the results of various mutual exclusion algorithms under the situation that (1) every involved site produces a critical section request randomly after its earlier request gets served, and (2) every site makes a large number of critical section requests (say hundreds or thousands of them) before encountering a barrier synchronization, if any. This situation exists in many real applications, such as the traveling salesman problem developed at Rice University [32], QuickSort (QS), and Water from the Stanford Parallel Applications for Shared Memory (SPLASH) benchmark suite [33]. However, there are other situations where a participating site makes a few requests to the critical section before involving one barrier synchronization. An example of these situations can be found in such a benchmark code as Integer Sort (IS) in the NAS benchmark suite [30]; specifically, a site in IS requests the critical section exactly once before facing a barrier synchronization. Under these situations, all involved sites after the barrier synchronization issue their critical section requests within a short period of time, causing high contention. The mutual exclusion algorithms under these situations are investigated here to unveil their behaviors.

The empirical study under these situations was conducted on the IBM SP2 machine with N equal to 64. Again, every involved site issues mutual exclusion

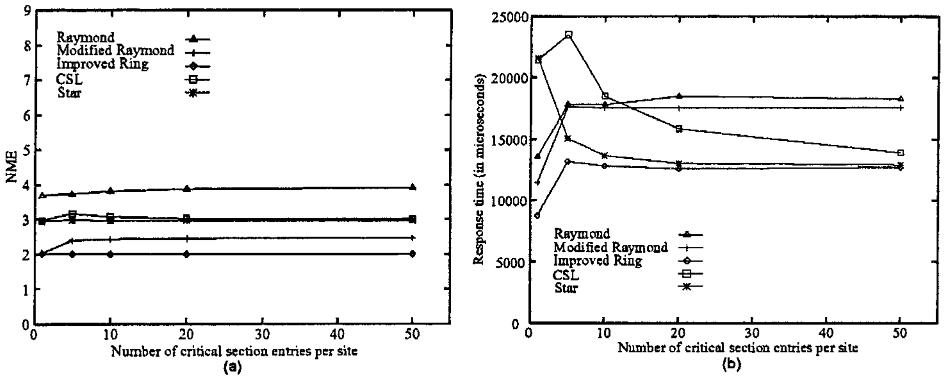

FIG. 9. NME and response time versus number of critical section entries achieved by each individual processor on SP2 with $N = 64$, under $\Gamma = 100\,\mu s$ and $\xi = 100\,\mu s$.

requests randomly with a mean value of Γ, which is the mean time between releasing the critical section and requesting it again by the site. We plot NME and response time versus number of critical section entries in Fig. 9, where Γ and ξ are both set to $100\,\mu s$ (to reflect an extremely high degree of contention). When the number of critical section entries is 1, the improved Ring algorithm is observed to yield the shortest response time and the smallest NME, because each site in the ring structure sends one message to its successor and all the sites are ready for the critical section under this Γ value. In contrast, every site in the Star algorithm sends its critical section request message to the root at about the same time (under this Γ value), causing serious contention and thus exhibiting the worst response time. Similarly, the CSL algorithm also suffers from high contention and results in a long response time.

If each site enters the critical section multiple times before encountering a barrier synchronization during the course of program execution, the degree of contention subsides gradually. This is because a site normally does not produce another request for the critical section while waiting for the earlier request to be served, and the sites enter then leave the critical section one at a time. While all sites start generating requests for the critical section within a short time period initially, they generate their second critical section requests over a much longer period of time, reducing the degree of contention. All sites eventually serialize their generation of requests for the critical section and almost avoid producing critical section requests at the same time. This phenomenon makes the Star algorithm quickly become the most efficient algorithm, as can be observed in Fig. 9b. These results reveal that the Star algorithm is preferred if each site requires the critical section to be centered a number of times before encountering any barrier synchronization.

The CSL algorithm has the same NME and response time as the Star algorithm when the number of critical section entries equals 1, because both of them have the same logical structure (i.e., a star) initially. As the program proceeds and reaches five critical section entries (see Fig. 9), the CSL algorithm exhibits a higher NME and response time because the token, under such a heavy request load, travels quite frequently among sites and the dynamic logical structure, at this moment, is under a transient stage (i.e., no longer a star) which cannot effectively adjust and evolve.

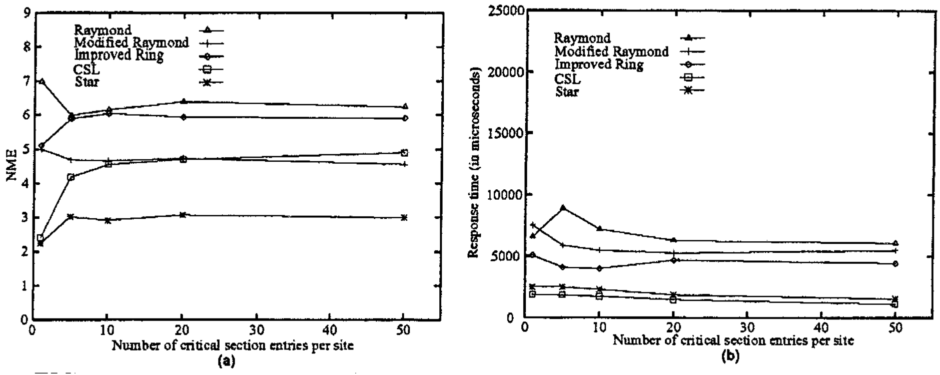

FIG. 10. NME and response time versus number of critical section entries achieved by each individual processor on SP2 with $N = 64$, under $\Gamma = 18{,}000\,\mu s$ and $\xi = 100\,\mu s$.

While entering the critical section many times, all sites in the CSL algorithm gradually serialize their generation of requests; meanwhile, the logical structure evolves to a relatively stable condition, leading to pronounced reduction in NME and response time as shown in Fig. 9.

The situation for a light request load of $\Gamma = 18{,}000\,\mu s$ is illustrated in Fig. 10, where the Star and CSL algorithms exhibit shorter response times than others for any given number of critical section entries due to extremely low contention.

5. PERTINENT WORK

A simulation study was performed previously by Johnson [6] to contrast mutual exclusion algorithms described in [7–9], mainly in terms of NME. That simulation study examined a different construct of the Star algorithm [8], with its logical structure changed dynamically, as opposed to a fixed structure with a designated site as the root in our implementation. When the root of the Star algorithm is fixed, it requires no more than three messages exchanged per critical section entry, exhibiting better performance than a construct with its structure changed dynamically. Raymond's algorithm studied in that simulation study is limited to its original form, but our work here introduces a modified version of the algorithm and compares the behaviors of both the original and the modified Raymond algorithms. We also present the results of our improved Ring algorithm.

Our empirical evaluation gives rise to a different conclusion than the earlier simulation study performed in [6] does, as stated in the following. The CSL and original Raymond algorithms are investigated by both studies. Raymond's algorithm, according to Johnson's simulation results [6], exhibits a lower NME than the CSL algorithm under heavy request loads, whereas our empirical study indicates that the CSL algorithm outperforms Raymond's one throughout all request loads investigated, including extremely heavy load. The main reasons behind this difference are that the simulation study in [6] (1) could not truly reflect real traffic patterns generated by these algorithms, (2) is unable to simulate the logical tree structure under real machines, (3) could not fully take into account the program behaviors (for example, maintaining the local queue and logical structures) that impact the performance, and (4) fails to mimic possible network congestion. By contrast, our empirical study addresses these issues through actual implementation of these algorithms on real machines, obtaining realistic outcomes. In fact, the NME values of Raymond's algorithm collected by our empirical work are quite close to the analytic result provided by Raymond's work [7] in that the best NME value (under extremely heavy load) is $\frac{4(N-1)}{N}$, signifying on an average approximately four messages involved in mutual exclusion under a system with N sites.

6. CONCLUDING REMARKS

We have investigated empirically various distributed mutual exclusion algorithms, including our proposed modified Raymond algorithm and improved Ring algorithm [10], in this study. Our investigation was conducted on the IBM SP2 machine and the Intel *iPSC*/860 system. The performance measures of interest are NME (i.e., mean number of messages exchanged per section entry) and response time. If sites produce requests to the critical section repetitively many times before involving any barrier synchronization, the Star algorithm achieves the best performance from the response time standpoint under almost all request loads, closely followed by the CSL algorithm, On the other hand, the improved Ring algorithm [10] is found to yield the smallest NME under a heavy request load, but the Star algorithm leads to the lowest NME under a moderate to light request rate. A larger critical section duration (ξ) reduces the NME value but increases the response time for all algorithms. In particular, the Ring algorithm and the (modified) Raymond algorithm experience more pronounced reduction in NME than the Star algorithm and the CSL algorithm when ξ grows. From the scalability standpoint, the Star algorithm and the CSL algorithm are superior, since they exhibit smaller NME values and response times than Raymond's algorithm and the Ring algorithm when the system size increases. Being simple yet effective for practical system sizes, the Star algorithm has been employed to implement mutual exclusion in distributed shared-memory systems under the lazy release consistency model [28].

If every site enters the critical section only once before facing a barrier synchronization, different conclusions result, depending on the request rate. Under a heavy request load, our improved Ring algorithm seems to be most attractive, as it results in the smallest NME and the shortest response time. For a light request load, however, the CSL and Star algorithms prevail. As a result, the best algorithm for mutual exclusion in distributed memory systems depends on how those involved sites produce mutual exclusion requests.

The experimental results presented here could be useful for the future distributed system design, like a computer cluster with its constituent machines interconnected by a network. In such a distributed system, critical section execution typically involves some operations that read from, or write to, remote memory locations. Those remote memory accesses must go through the network, thus prolonging the critical section duration. If the network for interconnection is made with reduced latency in the future, the critical section duration experienced by each site in such a cluster system shrinks, making the Star and CSL algorithms more attractive than others, as demonstrated by our experimental results shown in Figs. 5 and 6. Consequently, a faster network with lower latency for constructing distributed systems would favor the Star and CSL algorithms.

ACKNOWLEDGMENTS

We thank Argonne National Laboratory and ERC for Computational Field Simulation, Mississippi State University, for allowing our access to the IBM SP2 machine and the Intel iPSC/860 system, respectively. The IBM SP2 machine belongs to the Argonne High-Performance Computing Research Facility, which is funded principally by the U.S. Department of Energy, Mathematical, Information and Computational Sciences Division (ER-31). We also thank Zhiyuan Li for his suggestion of the modified Raymond algorithm.

REFERENCES

1. A. Kumar, Hierarchical quorum consensus: A new algorithm for managing replicated data, *IEEE Trans. Comput.* **40**, 9 (September 1991), 996–1004.
2. R. H. Thomas, A majority consensus approach to concurrency control for multiple copy databases, *ACM Trans. Database Systems* **4**, 2 (June 1979), 180–209.
3. M. Stumm and S. Zhou, Algorithms implementing distributed shared memory, *Computers* (May 1990), 54–64.
4. K. Gharachorloo *et al.*, Memory consistency and event ordering in scalable shared-memory multiprocessors, *in* "Proc. 17th Int. Symp. Computer Architecture," pp. 15–26, May 1990.
5. T. Agerwala *et al.*, SP2 system architecture, *IBM System J.* **34**, 2 (1995), 152–184.
6. T. Johnson, A performance comparison of fast distributed mutual exclusion algorithms, *in* "Proc. 9th Int. Parallel Processing Symp.," pp. 258–264, April 1995.
7. K. Raymond, A tree based algorithm for distributed mutual exclusion, *ACM Trans. Comput. Systems* **7**, 1 (February 1989), 61–77.
8. M. L. Neilsen and M. Mizuno, A dag-based algorithm for distributed mutual exclusion, *in* "Proc. 11th Int. Conf. Distributed Computing Systems," pp. 354–360, May 1991.
9. Y. I. Chang, M. Singhal, and M. T. Liu, An improved $O(\log N)$ mutual exclusion algorithm for distributed systems, *in* "Proc. 1990 Int. Conf. Parallel Processing," Vol. III, pp. 295–302, August 1990.
10. S. S. Fu and N.-F. Tzeng, "Efficient Token-Based Approach to Mutual Exclusion in Distributed Memory Systems," Tech. Rep. TR-95-8-1 CACS, Univ. Southwestern Louisiana, Lafayette, LA, July 1995.
11. L. Lamport, Time clocks and ordering of events in distributed systems, *Commun. ACM* **21**, 7 (July 1978), 558–565.
12. G. Ricart and A. Agrawala, An optimal algorithm for mutual exclusion in computer networks, *Commun. ACM* **34**, 2 (January 1981), 9–17.
13. O. S. F. Carvallo and G. Roucairol, On mutual exclusion in computer networks, *Commun. ACM* **26**, 2 (February 1983), 146–147.
14. M. Singhal, A dynamic information-structure mutual exclusion algorithm for distributed systems, *IEEE Trans. Parallel Distrib. Systems* **3**, 1 (January 1992), 121–125.
15. M. Maekawa, A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems, *ACM Trans. Comput. Systems* **3**, 2 (May 1985), 145–159.
16. D. Agrawal and A. E. Abbadi, An efficient and fault-tolerant solution for distributed mutual exclusion, *ACM Trans. Comput. Systems* **9**, 1 (February 1991), 1–20.

17. T. Ibaraki and T. Kameda, A theory of coteries: Mutual exclusion in distributed systems, *IEEE Trans. Comput.* **4**, 7 (July 1993), 779–794.

18. G. LeLann, Algorithms for distributed data-sharing systems which use tickets, *in* "Proc. 3rd Berkeley Workshop Distributed Data Management and Computer Networks," pp. 259–272, 1978.

19. M. Naimi and M. Trehel, An improvement of the log n distributed algorithm for mutual exclusion, *in* "Proc. 17th Int. Conf. Distributed Computing Systems, W. Berlin, Germany, September 1987," pp. 371–375.

20. I. Suzuki and T. Kasami, A distributed mutual exclusion algorithm, *ACM Trans. Comput. Systems* **3**, 4 (November 1985), 344–349.

21. M. Singhal, A heuristically-aided algorithm for mutual exclusion in distributed systems, *IEEE Trans. Computers* **38**, 5 (May 1989), 651–662.

22. K. Makki, An efficient token-based distributed mutual exclusion algorithm, *J. Comput. Software Engineering* (December 1994).

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

24. A. J. Martin, Distributed mutual exclusion on a ring of processes, *Sci. Comput. Progr.* **5** (February 1985), 265–276.

25. "Document for a Standard Message-Passing Interface," Message Passing Interface Forum, Tech. Rep., CS-93-214 Univ. Tennessee, November 1993.

26. Z. Xu and K. Hwang, Modeling communication overhead: MPI and MPL performance on the IBM SP2, *IEEE Parallel Distrib. Tech.* (Spring 1996), 9–23.

27. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms," pp. 505–509, MIT Press, Cambridge, MA, 1990.

28. P. Keleher *et al.*, Lazy release consistency for software distributed shared memory, *in* "Proc. 19th Int. Symp. Computer Architecture," pp. 13–21, May 1992.

29. S. Venugopal *et al.*, Performance of distributed sparse Cholesky factorization with prescheduling, *in* "Proc. Supercomputing '92," pp. 52–61, Nov. 1992.

30. D. Bailey *et al.*, "The NAS Parallel Benchmarks," Tech. Rep. TR RNR-91-002 NASA Ames, August 1991.

31. X. Zhang *et al.*, Evaluating and designing software mutual exclusion algorithms on shared-memory multiprocessors, *IEEE Parallel Distrib. Tech.* (Spring 1996), 25–42.

32. P. Keleher, "Distributed Shared Memory Using Lazy Release Consistency," Ph.D. thesis, Rice University, December 1994.

33. J. P. Singh *et al.*, "SPLASH: Stanford Parallel Applications for Shared-Memory," Tech. Rep. CSL-TR-91-469, Stanford University, April 1991.

34. S. Banerjee and P. K. Chrysanthis, A new token passing distributed mutual exclusion algorithm, *in* "Proc. 16th Int. Conf. Distributed Computing Systems," pp. 717–724, May 1996.

35. M. Mizuno, M. Nesterenko, and H. Kakugawa, Lock-based self-stabilizing distributed mutual exclusion algorithms, *in* "Proc. 16th Int. Conf. Distributed Computing Systems," pp. 708–716, May 1996.

36. Y. I. Chang, A simulation study on distributed mutual exclusion, *J. Parallel Distrib. Comput.* **33**, 2 (March 1996), 107–121.

37. J. M. Helary, A. Mostefaoui, and M. Raynal, A general scheme for token- and tree-based distributed mutual exclusion algorithms, *IEEE Trans. Parallel Distrib. Systems* **5**, 11 (November 1994), 1185–1196.

SHIWA S. FU received a B.S. in information and computer engineering in 1986 from the Chung-Yuan University, Taiwan, an M.S. in computer science in 1990 from the State University of New York at Binghamton, and a Ph.D. in computer science in 1997 from the University of Louisiana at Lafayette. Prior to entering SUNY-Binghamton, he was a System Engineer with WANG Industrial Corp., Taiwan, and an Assistant Engineer with Industrial Technology Research Institute (ITRI), Taiwan. Since June

1997, he has been with IBM Thomas J. Watson Research Center, Hawthorne, New York. His current research interest is in the areas of electronic commerce, parallel processing, distributed computing, and operating systems.

NIAN-FENG TZENG received the Ph.D. in computer science from the University of Illinois at Urbana–Champaign in 1986. He has been with the Center for Advanced Computer Studies, University of Louisiana at Lafayette since 1987. His current research interest is in the areas of high-performance computer systems, parallel and distributed processing, high-speed networking, and fault-tolerant computing. He is on the editorial board of the IEEE Transactions on Parallel and Distributed Systems and was on the editorial board of the IEEE Transactions on Computers, 1994–1998. He served as Co-Guest Editor of a special issue of the *Journal of Parallel and Distributed Computing* on distributed shared memory systems, September 1995, and as a Distinguished Visitor of the IEEE Computer Society, 1994–1997. He is the Chair of the Technical Committee on Distributed Processing, the IEEE Computer Society, and has been on the technical program committees of various conferences. Dr. Tzeng is a senior member of the IEEE, a member of the Association for Computing Machinery, and the recipient of the outstanding paper award of the 10th International Conference on Distributed Computing Systems, May 1990. He received the University Foundation Distinguished Professor Award in 1997.

JEN-YAO CHUNG received the B.S. in computer science and information engineering from National Taiwan University, 1982, and the M.S. and Ph.D. in computer science from the University of Illinois at Urbana–Champaign, 1989. Since June 1989, he has been with IBM Thomas J. Watson Research Center, Hawthorne, New York, as a research staff member. He currently is the manager of the electronic commerce architecture and manager of the IBM Institute of Advanced Commerce Technology Organization. He has authored and co-authored over 50 technical papers in journals and conferences. He has been involved in the research and development of electronic commerce and web application systems. Areas of current research are XML/EDI, integrating existing business with WWW, backend ERP integration, web data access, and scalable web servers. He was awarded the IEEE Outstanding Paper Award in 1995, the IBM Outstanding Technical Achievement Award in 1994, the IBM Outstanding Contribution Award in 1997, and two Research Division Awards in 1990 and 1996. He is a senior member of IEEE and a member of ACM.