

# **Fast Mutual Exclusion Algorithms**

## **The MPI Implementation**

Andrew E. Santosa (HD99-3028J)

`andrews@comp.nus.edu.sg`

23 October 2000

This report has been submitted for completing a coursework assignment for CS4231: Parallel and Distributed Algorithm of the School of Computing, National University of Singapore.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Algorithms</b>	<b>2</b>
2.1	Lamport's Algorithm . . . . .	2
2.2	Alur and Taubenfeld's Algorithm . . . . .	3
2.3	Michael and Scott's Algorithm . . . . .	5
2.4	Styer's Algorithm . . . . .	6
2.5	MCS Lock . . . . .	8
2.6	Huang's Algorithm . . . . .	11
<b>3</b>	<b>MPI Implementation</b>	<b>14</b>
3.1	Lamport's Algorithm . . . . .	16
3.2	Alur and Taubenfeld's Algorithm . . . . .	19
3.3	Michael and Scott's Algorithm . . . . .	22
3.4	Styer's Algorithm . . . . .	24
3.5	MCS Lock . . . . .	30
3.6	Huang's Algorithm . . . . .	33
<b>4</b>	<b>Comparisons</b>	<b>36</b>
<b>5</b>	<b>Conclusion</b>	<b>37</b>

# 1 Introduction

A mutual exclusion algorithm is *fast* if a processor enters the critical section within a constant number of steps when it is the only processor trying to enter the critical section [2].

This paper intends to implement various fast mutual exclusion algorithms proposed to date. It starts from Lamport's algorithm (1987) to Huang's algorithm (1998). In this report we concentrate on distributed systems, hence we neglect, for example, fast mutual exclusion for uniprocessor systems by Bershada et al. [3].

Implementations are done using the Message Passing Interface (MPI) [5].

In Section 2 we provide pseudocode and informal explanation of each algorithm. In Section 3 we provide explanation on the MPI implementation of each algorithm. In Section 4 we provide comparisons based on actual output results of MPI implementation, and finally we conclude our report in Section 5.

## 2 Algorithms

In presenting the pseudocodes in this section, we use the convention used in [2]. Most importantly:

- Variables are printed in italic. Shared variable names begin with a capital letter, while local variable names consist of all lowercase characters.
- Atomic procedure labels are printed in serif.
- For read/write variables, we use assignment statement, e.g.,  $A := b$ . This statement reads value from  $b$  and stores it in  $A$ .

### 2.1 Lamport's Algorithm

Lamport described that the sequence of instructions that must be executed before a processor can enter its critical section is *write X, read y, write y, read x*, where  $x$  and  $y$  are distinct variables in the shared memory, based on informal arguments [9]. This idea is manifested in the first algorithm of Figure 1.

The delay at line 6 must be long enough for any processor that has already read  $Slow-lock = -1$  at line 2 to complete lines 4, 5 and, if appropriate, the critical section and 9.

The algorithm in Figure 1 is based on assumption on upper bound on memory access time (the use of delay in line 6). Also, A processor that has entered the critical section through the *Fast-lock* cannot tell other processors when it has completed the critical section. The algorithm thus assumes an upper bound on the execution of critical section as well. An improved algorithm is shown in Figure 2.

Code to be executed by processor  $p_i$ ,  $0 \leq i \leq n - 1$ .  
Initially, *Slow-lock* is -1.

```
⟨Entry⟩:  
1: Fast-lock :=  $i$   
2: if Slow-lock  $\neq$  -1 then  
3:   goto 1  
4: Slow-lock :=  $i$   
5: if Fast-lock  $\neq$   $i$  then  
6:   delay  
7:   if Slow-lock  $\neq$   $i$  then  
8:     goto 1  
⟨Critical Section⟩  
⟨Exit⟩:  
9: Slow-lock := -1  
⟨Remainder⟩
```

Figure 1: Lamport's First Algorithm

## 2.2 Alur and Taubenfeld's Algorithm

This is an improvement over Lamport's algorithm for *fast* mutual exclusion. An algorithm is fast if it allows a process to enter the critical section in  $(O(1))$  time when there is no contention. The main argument is that Lamport's algorithm needs to check the status of all participating process when there is contention. This status checking necessitate excessive shared memory references. Instead of checking status of participants, Alur and Teubenfeld introduces delay to allow all processes competing for critical section to come to a stable state before further checking shared memory status. The delay assumes an upper bound on the time required for shared memory reference. This assumption is allowed when the system is embedded or with hardware interrupts disabled [11]. This algorithm is slightly improved by Michael and Scott [11].

Alur and Taubenfeld tried to combine Lamport's and Fischer's algorithm. Fischer's algorithm is shown in Figure 3. Fischer's algorithm is based on assumption on upper bound on memory access time. The delay in line 4 is so that if another process had found that *Lock* = -1 before  $p_i$  sets *Lock* to  $i$ , the other process would have set its own id to *Lock*. The disadvantage of Fischer's algorithm is that even in the absence of contention a processor needs to delay itself.

On the other hand, Lamport's fast algorithm has its own drawback as well. In line 10 of Figure 2, there is a need to check the status of all other participating processors if a processor uses the *Slow-lock*, no matter how small the contention is.

Instead of doing this, Alur and Taubenfeld proposed the use of delay, as in

Code to be executed by processor  $p_i$ ,  $0 \leq i \leq n - 1$ .  
 Initially, *Fast-lock* and *Slow-lock* are -1, and  
*Want*[ $i$ ] is false for all  $i$ ,  $0 \leq i \leq n - 1$ .

```

⟨Entry⟩:
1:  Want[ $i$ ] := true
2:  Fast-lock :=  $i$ 
3:  if Slow-lock  $\neq$  -1 then
4:    Want[ $i$ ] := false
5:    wait until Slow-lock = -1
6:    goto 1
7:  Slow-lock :=  $i$ 
8:  if Fast-lock  $\neq$   $i$  then
9:    Want[ $i$ ] := false
10:  for all  $j$ , wait until Want[ $j$ ] = false
11:  if Slow-lock  $\neq$   $i$  then
12:    wait until Slow-lock = -1
13:    goto 1
⟨Critical Section⟩
⟨Exit⟩:
14: Slow-lock := -1
15: Want[ $i$ ] := false
⟨Remainder⟩

```

Figure 2: Lamport's Second Algorithm

Code executed by processor  $p_i$ ,  $0 \leq i \leq n - 1$ .  
 Initially, *Lock* = -1.

```

⟨Entry⟩:
1:  repeat
2:    wait until Lock = -1
3:    Lock :=  $i$ 
4:    delay
5:  until  $x = i$ 
⟨Critical Section⟩
⟨Exit⟩:
6:  Lock := 0
⟨Remainder⟩

```

Figure 3: Fischer's Algorithm

Code to be executed by processor  $p_i$ ,  $0 \leq i \leq n - 1$ .  
Initially, *Slow-lock* and *Z* are both -1 and false,  
respectively.

```
⟨Entry⟩:
1:  Fast-lock := i
2:  wait until Slow-lock = -1
3:  Slow-lock := i
4:  if Fast-lock ≠ i then
5:    delay
6:    if Slow-lock ≠ i then
7:      goto 1
8:    wait until Z = false
9:  else
10:   Z := true
⟨Critical Section⟩
⟨Exit⟩:
11: Z := false
12: if Slow-lock = i then
13:   Slow-lock := -1
⟨Remainder⟩
```

Figure 4: Alur and Taubenfeld's Algorithm

Fischer's algorithm. Alur and Taubenfeld's algorithm is shown in Figure 4. The delay at line 5 is assumed to be long enough for any processor that has already read *Slow-lock* = -1 in line 2 to complete line 3, and any process that has already set *Slow-lock* in line 3 to complete line 4 and (if not delayed) line 10 (that is the time required to service 2 memory references by all other processors in the system) [1].

### 2.3 Michael and Scott's Algorithm

This report describes a little improvement over Alur and Taubenfeld's algorithm for fast mutual exclusion [1]. Alur and Taubenfeld's algorithm uses two shared variables, say *fast-lock* and *slow-lock*, in which in no contention a process enters the critical section by setting the *fast-lock* variable, and when there is contention, it enters the critical section by setting the *slow-lock* variable. In order for the competing processes to know whether there is still a process in the critical section, a special variable *Z* is introduced. Michael and Scott's algorithm basically allows the referring of both *Z* and *slow-lock* variables in one shot, therefore reducing the number of messages.

Michael and Scott's algorithm is another improvement on Lamport's algorithm [11]. It is based on the same assumption as Alur and Taufenbeld's regarding

Code to be executed by processor  $p_i$ ,  $0 \leq i \leq n - 1$ .  
Initially,  $Slow-lock = -1$ ,  $F = out$ .

```

⟨Entry⟩:
1:  Fast-lock := i
2:  if Slow-lock ≠ -1 then
3:    goto 1
4:  Slow-lock := i
5:  if Fast-lock ≠ i then
6:    delay
7:    if (Slow-lock, F) ≠ (i, out) then
8:      goto 1
9:  F := in
⟨Critical Section⟩
⟨Exit⟩:
10: (Slow-lock, F) := (-1, out)
⟨Remainder⟩

```

Figure 5: Michael and Scott’s Algorithm

upper bound on message passing time. It has a slight improvement over Alur and Taubenfeld’s due to less shared memory reference, although of the same  $O(1)$  when there is no contention. At contention, shared memory reference in both algorithm is  $O(n)$ .

Michael and Scott’s algorithm uses  $F$  variable to replace  $Z$ . A tuple of  $Slow-lock$  and  $F$  can be read and written from the shared memory at once in an atomic operation.

The delay at line 6 is assumed to be long enough for any process that has already read  $Slow-lock = -1$  in line 2 to complete line 4, and any process that has already set  $Slow-lock$  in line 4 to complete line 5, and, if not delayed, line 9. The amount of delay is thus the same as the time required to service 2 memory references by all other processors in the system.

## 2.4 Styer’s Algorithm

Styer proposed an improvement over Lamport’s algorithm [12]. If there was no contention, then a process may enter the critical section in  $l + 7$  operations, where  $l$  is the number of processes allowed to execute their critical sections in parallel. If  $l = 1$ , then  $O(\log t)$  is all that is required in lock acquisition when there is contention, where  $t$  is the number of competing processes.

Styer’s algorithm acts as a wrapper to “normal” mutual exclusion algorithm (e.g., usual “slow” mutual exclusion algorithm— $O(n)$  operations although there is no contention).

```

⟨Entry⟩:
1:   $W_i := \text{Fast}$ 
2:  if  $Lock = \text{true}$  then
3:     $W_i = \text{Slow}$ 
4:    for  $j := 0$  to  $l - 1$  do
5:      entry( $j, \text{sub}(i, j + 1), \text{sub}(i, j)$ );
6:       $Block := \text{true}$ 
7:      check_vars( $l - 1, 0$ )
8:    else
9:      for  $j := 1$  to  $l - 1$  do
10:        $S_{j, \text{sub}(j, i)} := i$ 
11:        $Lock := \text{true}$ 
12:       if  $Turn \neq i$  or  $Block = \text{true}$  then
13:         goto 3
⟨Critical Section⟩
⟨Exit⟩:
14:  $Lock := \text{false}$ 
15: if  $W_i = \text{Fast}$  then
16:   $W_i := \text{Out}$ 
17: else
18:  Block := false
19:   $W_i := \text{Out}$ 
20:  for  $j := l - 1$  downto 0 do
21:    Exit( $j, \text{sub}(i, j + 1), \text{sub}(i, j)$ )
⟨Remainder⟩

```

Figure 6: Styer's Algorithm

Initially  $V$  equals 0.

```

⟨Entry⟩: wait until test&set( $V$ ) = 0
⟨Critical Section⟩
⟨Exit⟩:
2: reset( $V$ )
⟨Remainder⟩

```

Figure 7: Mutual Exclusion Using Test-and-Set Register

```

check_vars(level: unsigned integer, start: unsigned integer):
  if level > 0 then
    for j := 0 to k - 1 do
      wait until  $W_{S_{level, start+j}} \neq \text{Fast}$ 
      if  $W_{S_{level, start+j}} = \text{Slow}$  then
        check_vars(level - 1, (start + j) × k)
    else
      for j := 0 to k - 1 do
        wait until  $W_{start+j} \neq \text{Fast}$ 

```

As the normal mutual exclusion algorithm, here we will use the simple mutual exclusion algorithm based on test-and-set register. Here, we assume these operations on the shared memory as atomic:

test&set( $V$ : memory address) returns binary value:

```

temp := V
V := 1
return (temp)

```

reset( $V$ : memory address):

```

V := 0

```

The test&set operations has one memory address argument. It sets the value of the memory address to 1, and returns the old value contained in the memory address. The reset operation simple sets the value of the memory address given as argument to 0.

The mutual exclusion algorithm with test&set register is shown in Figure 7. In line 1, a processor simply wait for the test&set register to become 0, which means that there is no other processor is in its critical section. When exiting from the critical section, a processor sets the register to 0.

## 2.5 MCS Lock

Mellor-Crummey and Scott invented the MCS lock [10] which has the following characteristics:

- Guarantees FIFO ordering of lock acquisitions. FIFO guarantees non-lockout, which further implies non-deadlock.
- Spins on locally-accessible flag variables only. This can be expected to increase performance compared with traditional algorithms that spins on globally-accessible variable.
- Requires a small constant amount of space per lock.
- It is also fast, i.e., requiring only  $O(1)$  network transactions per lock acquisition.

We assume a record data type `qnode` as below:

```
type qnode = record
  next: *qnode
  locked: Boolean
```

We also define a lock data type, which is a pointer to `qnode` as follows:

```
type lock = *qnode
```

Here we make some addition to syntax convention defined in [2]. `*` means pointer as in C [8]. We refer to an element of a record type using infix binary “ $\rightarrow$ ” as in  $I \rightarrow next$ , in which the first argument ( $I$  in our example) is a pointer.

Here there are two notions of shared memory:

- *Locally-accessible* shared memory is the shared memory that is less costly when accessed from particular processor  $p_i$  than from other processors.
- *Globally-accessible* shared memory has the same access cost by all processors, and the cost is maximum in the system.

On spin-based algorithms, it is therefore desirable for processor  $p_i$  to spin on its locally-accessible shared memory than globally-accessible shared memory.

The algorithm is based on two atomic operations:

- `Fetch&store` takes two arguments. The first argument is a pointer to the lock in globally-accessible shared memory, and the second argument is the local `qnode` on a processor. `Fetch&store` replace the pointer value of the lock with the local `qnode`, and return the old value of the lock.

```
fetch&store(A: memory address, B: memory address)
  returns binary value:
  temp := A
  A := B
  return (temp)
```

$I$  points to a qnode record allocated in shared memory locally-accessible to the invoking processor. Initially,  $I \rightarrow next = nil$ , and  $L$  is also of type qnode, but located in globally-accessible shared memory.

```

⟨Entry⟩:
1:   $I \rightarrow next := nil$ 
2:   $predecessor := \text{fetch\&store}(L, I)$ 
3:  if  $predecessor \neq nil$  then
4:     $I \rightarrow locked := true$ 
5:     $predecessor \rightarrow next := I$ 
6:    wait until  $I \rightarrow locked = false$ 
⟨Critical Section⟩
⟨Exit⟩:
7:  if  $I \rightarrow next = nil$ 
8:    if  $\text{compare\&swap}(L, I, nil)$  then
9:      goto ⟨Remainder⟩
10:  wait until  $I \rightarrow next \neq nil$ 
11:  $I \rightarrow next \rightarrow locked := false$ 
⟨Remainder⟩

```

Figure 8: MCS Lock

- Compare&swap takes three arguments. It compares the first with the second, and if both are the same, the first's value is replaced with the third's and true is returned. Otherwise, it returns false.

```

compare&swap( $A$ : memory address,  $B$ : memory address,
              $C$ : memory address) returns binary value:
   $temp := A$ 
  if  $temp = B$  then
     $A := C$ 
  return ( $temp$ )

```

The pseudocode of MCS lock is shown in Figure 8.

The entry part can be explained as follows: A processor  $p_i$  upon trying to enter its critical section, uses `fetch&store` to have the lock  $L$  points to its qnode  $I$ , and saves the qnode previously pointed by  $L$  in local variable `predecessor`. If a predecessor actually exists, after locking itself (line 4) and making the predecessor points to its qnode (line 5), it starts spinning (line 6), waiting for the predecessor to signal. Otherwise, it enters its critical section.

Note that since only the direct predecessor can signal a blocking process, it is clear that this algorithm ensures FIFO property. It is also clear that the algorithm

is fast since lock acquisition is of constant network transactions.

In the exit part, the processor exiting its critical section first inspects the existence of *next* qnode (line 7). If there is one, that qnode is signaled to stop spinning (line 11). Otherwise, the exiting processor checks whether a *fetch&store* has been executed by another processor (line 8). If yes, in which case *compare&swap* returns the old value of *L*, the processor will spin wait for another processor have it set its *next* that is later used to unlock the other processor (line 11).

There is a drawback in this algorithm in that the last processor executing the exit section when there is no more processor is interested in entering the critical section should wait for another processor to execute the entry section. This is because the *compare&swap* of line 8 will return its own qnode, and the processor must spin at line 10. The algorithm thus cannot guarantee fairness in the exit section. This explains why our implementation described in Section 3.5 does not terminate.

## 2.6 Huang's Algorithm

Huang's works are improvement of Fu and Tzeng's circular list-based mutual exclusion scheme (CL algorithm) [4]. CL algorithm is similar to MCS lock in which processors spin on locally-accessible shared memory variable. Huang devised a race condition and starvation errors in CL algorithm in [7], and proposed several improvements in [6]. The algorithms presented here are all based on Huang's work in [6].

Huang tried to minimize the *linked list redirection problem* in MCS lock. Linked list redirection problem is access of shared variable in other processor's local memory (e.g., line 5 and 11 of Figure 8).

Shown in Figure 9 is Huang's first algorithm. Although in his paper Huang used flowchart, here we represent the algorithms using usual pseudocode convention. This is done by semantics-preserving transformation.

The first algorithm uses *fetch&store* and *swap&compare* primitives. The definition of *fetch&store* is the same as in MCS lock. On the other hand, *swap&compare* is defined as follows:

```
swap&compare(A: memory address, B: memory address,  
             C: memory address):  
  temp := A  
  A := B  
  B := temp  
  if A = B then  
    A := C
```

The definition of *swap&compare* was first appeared in Fu and Tzeng's paper [4].

The explanation of Huang's algorithm is as follows: In line 1, the algorithm first uses *fetch&store* to store its qnode in *L* and stores old *L* value in a local variable *next*. If there has been a processor interested in entering the critical section before

Initially,  $L = \text{nil}$ , at each processor  $p_i$ ,  $I \rightarrow \text{wait} = \text{true}$ .

```
⟨Entry⟩:
1:   $next := \text{fetch\&store}(L, I)$ 
2:  if  $next \neq \text{nil}$  then
3:    wait until  $I \rightarrow \text{wait} = \text{false}$ 
⟨Critical Section⟩
⟨Exit⟩:
4:  if  $next = \text{nil}$  then
5:    loop
6:     $next := I$ 
7:     $\text{swap\&compare}(L, next, \text{nil})$ 
8:    if  $next = I$  then
9:      goto ⟨Remainder⟩
10:    $I \rightarrow \text{wait} := \text{true}$ 
11:    $next \rightarrow \text{wait} := \text{false}$ 
12:   wait until  $I \rightarrow \text{wait} = \text{false}$ 
13:    $next \rightarrow \text{wait} = \text{false}$ 
⟨Remainder⟩
```

Figure 9: Huang's First Algorithm

$p_i$ , in which case  $next$  is non-nil, the processor will suspend in line 3 waiting to be waken up. Otherwise,  $p_i$  will enter the critical section.

It is clear that this algorithm is fast, since a processor can enter its critical section in  $O(1)$  time provided there is no contention.

After completing the critical section, if  $next$  is non-nil (other processor is waiting), then  $p_i$  will simply wake the next waiting processor up in line 13. Otherwise, although  $next$  is nil (as it is returned by `fetch&store` in line 1, it is possible that some processors are already interested in entering the critical section and executed `fetch&store`, therefore making  $L$  not equal to  $I$ ). These processors, which have been waiting for  $p_i$  (in line 3), must be waken up. `Swap&compare` in line 7 is used to load the current value of  $L$  into local variable  $next$ , while setting  $L$  to  $p_i$ 's  $I$ . If it is found that the old value of  $L$ , which is now contained in  $next$  is equal to  $I$  (line 8), then we know that there is no other processor waiting for  $p_i$ , and  $p_i$  moves to remainder section (line 9). Otherwise,  $p_i$  will signal the waiting processor and  $p_i$  itself will spin in line 12. Here,  $p_i$  will act as a *controller*. It will be waken up by processors finishing critical section to wake up other waiting processors, hence the loop in line 5.

A drawback of the first algorithm is the possibility of starvation since  $p_i$  will not be able to leave its exit section when requests to enter the critical section keep arriving.

A second algorithm is shown in Figure 10. This algorithm basically allows a

Initially,  $L = \text{nil}$ , at each processor  $p_i$ ,  $I \rightarrow \text{wait} = \text{true}$ .

```

⟨Entry⟩:
1:   $next := \text{fetch\&store}(L, I)$ 
2:  if  $next \neq \text{nil}$  then
3:    wait until  $I \rightarrow \text{wait} = \text{false}$ 
⟨Critical Section⟩
⟨Exit⟩:
4:  if  $next = \text{nil}$  then
5:     $next := \text{compare\&swap}(L, I, \text{nil})$ 
6:    if  $next \neq I$  then
7:       $I \rightarrow \text{wait} := \text{true}$ 
8:       $next \rightarrow \text{direct} := \text{true}$ 
9:       $next \rightarrow \text{wait} := \text{false}$ 
10:   wait until  $I \rightarrow \text{wait} = \text{false}$ 
11:    $next \rightarrow \text{hold} := \text{false}$ 
12: else
13:    $next \rightarrow \text{wait} := \text{false}$ 
14:   if  $I \rightarrow \text{direct} = \text{true}$  then
15:     wait until  $I \rightarrow \text{hold} = \text{false}$ 
16:     goto 5
⟨Remainder⟩

```

Figure 10: Huang's Second Algorithm

controller to pass the its responsibility to the next processor waiting in line. This is possible because the controller sets the *direct* member of qnode structure of the next processor to true in line 8. The next processor detects whether it has directed to become a controller or not in line 14. If it is the case, then it will synchronize with the earlier controller before undertaking the job (the wait on  $I \rightarrow \text{hold}$  to become true in line 15).

Figure 11 shows the last algorithm. In this algorithm, to further reduce memory access, a *permission word* is passed in each remote memory write instead of a single bit. The permission word consists of head and tail halfwords, each being qnode address. Assuming the least significant bit (LSB) in memory addresses is not used for memory reference, the algorithm makes use of the LSB of  $p_i$ 's qnode address ( $I$ ) as *incarnation bit*. Incarnation bit is inversed at each attempt of a processor to compete for entering critical section.

Huang informally proved an impossibility result in [6] that there exists no algorithm that require less remote access than this algorithm does.

We explain the algorithm using an example: Assume a processor with  $I = 30+$  (+ means the incarnation bit is set) is first to enter the critical section. Since there is no other processor preceding it, it gets  $next = \text{nil}$  in line 1, and continue directly

to the critical section.

After the critical section, it executes the compare&swap at line 5. It is assumed that processors with  $I = 10+$ ,  $50+$ , and  $40+$  have attempted to enter the critical section, and have successfully executed fetch&store in that order<sup>1</sup>. Since all processors observed that  $next \neq nil$ , all spins at line 3. Therefore,  $L = 40+$ , and  $next$  is set to  $40+$  as the result of compare&swap. Since  $next \neq I$ , line 7 is executed. As the result, the permission word  $(30+,40+)$  is sent to the processor with  $I = 40+$ . Note that a processor that executes the compare&swap statement is called a *controller*.

The processor with  $I = 40+$  will thus terminate its spin at line 3, since now its permission word is not anymore 0, as set initially. The processor will thus enter its critical section. Since  $next = 50+$ , which is not equal to the head of permission word  $(30+,40+)$ , the permission word is simply passed to the next waiting process (with  $I = 50+$ ) in line 12. The processor now enters its remainder section.

Being passed the permission word, the processor with  $I = 50+$  stops its spin at line 3, and after performing similar procedure as its predecessor, passed the permission word  $(30+,40+)$  to the next processor (with  $I = 10+$ ).

The processor with  $I = 10+$  will have  $next = 30+$ , since it was the first processor that successfully executed fetch&store after the initial process with  $I = 30+$ . After being passed the permission word  $(30+,40+)$ , it stops spinning at line 3, and enter the critical section. After the critical section, the processor finds that  $next$  has the same value as the head of the permission word (line 9). Therefore, it sets its own word  $I$  to the tail of the permission word in line 10, which is  $40+$ . It then becomes the controller and executes the compare&swap in line 5.

As with previous controller (with  $I = 30+$ ), during the time some processors, for example the processor with  $I = 40-$ , may have been interested in entering the critical section. Processors with  $I = 40+$  and  $I = 40-$  are the same, but with the incarnation bit, the compare&swap will be able to distinguish between different attempts to enter critical section. Since  $40+ \neq 40-$ , line 7 is executed which result the permission word  $(40+,40-)$  is sent to processor with  $I = 40$ .

If, during the execution of mutual exclusion algorithm by the first run (with processors of  $I = 40+$ ,  $50+$ , and  $10+$ ), there was no other processor interested in the critical section, the compare&swap thus results in  $next = 40+$ , which is the same as current value of  $I$ , and which makes the processor simply moves to its remainder section.

### 3 MPI Implementation

We implemented the shared memory as managed using a server processor. In actual MPI implementation, the server processor has the rank 0. The server's processor's task is to serve requests to read/write variables.

---

<sup>1</sup>These are processors that are members of a *run*. Note that in a run, there cannot be multiple attempts by one processor, i.e., same  $I$  values (even when incarnation bit is different).

Initially,  $L = \text{nil}$ ,  $I$  is an address of  $p_i$ 's permission word,  $*I = 0$ , and the incarnation bit of  $I$  is the inverse of previous incarnation bit.

```
<Entry>:
1:   $next := \text{fetch\&store}(L, I)$ 
2:  if  $next \neq \text{nil}$  then
3:    wait until  $*I \neq 0$ 
<Critical Section>
<Exit>:
4:  if  $next = \text{nil}$  then
5:     $next := \text{compare\&swap}(L, I, \text{nil})$ 
6:    if  $next \neq I$  then
7:       $*next := \text{pack}(I, next)$ 
8:  else
9:    if  $next = \text{head}(*I)$  then
10:      $I := \text{tail}(*I)$ 
11:     goto 5
12:    $*next := *I$ 
<Remainder>
```

Figure 11: Huang's Third Algorithm

It is important to note that the server knows nothing about the client’s algorithm. The server is basically just a “dumb” program that serves read/write requests without reasoning about the contents. This is to maintain a clean separation of concerns.

In all cases, each client iterate for a specified number of time, attempting to enter the critical region at each iteration. Except for an exception in MCS lock as explained in Section 3.5, each client will eventually ends its loop and sends a notification to the server before it dies. Upon receiving notification from a client, the server reduces the number of live clients until it knows that there are no more clients, thus terminating itself.

### 3.1 Lamport’s Algorithm

It is important to note here that we used a 32-bit integer to replace the *Want* variable of Figure 2. Executing  $Want[i] = true$ , for example, would become setting on of bit  $i$  of integer *Want*. This proves that access to *Want* in line 10 of Figure 2 does not necessarily be costly, provided processor number is small (less than 32 for 32-bit).

```

1: /*
2:    Lamport’s Fast Mutual Exclusion Algorithm
3:    */
4:
5: #include <stdio.h>
6: #include "mpi.h"
7:
8: /* Message tags */
9: #define READ_FAST 50
10: #define WRITE_FAST 51
11: #define READ_SLOW 52
12: #define WRITE_SLOW 53
13: #define READ_WANT 54
14: #define SET_WANT 55
15: #define RESET_WANT 56
16: #define QUIT 57
17:
18: main(int argc, char** argv) {
19:     int my_rank; /* Rank of processor */
20:     int n; /* No. of processors */
21:     int i;
22:     int message; /* Storage for the message */
23:     MPI_Status status; /* Return status for the receive */
24:
25:     MPI_Init(&argc, &argv);
26:     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
27:     MPI_Comm_size(MPI_COMM_WORLD, &n);
28:
29:     if (!my_rank) {
30:         /* The shared memory server part */
31:
32:         int Fast_lock = -1;
33:         int Slow_lock = -1;
34:         int Want = 0; /* Here we implement want list as
35:                        integer instead of array */
36:

```

```

37:     for (;;) {
38:         MPI_Recv(&message, 1, MPI_INT, MPI_ANY_SOURCE,
39:                 MPI_ANY_TAG, MPI_COMM_WORLD, &status);
40:         switch (status.MPI_TAG) {
41:             case SET_WANT:
42:                 Want |= (1 << (status.MPI_SOURCE - 1));
43:                 break;
44:             case RESET_WANT:
45:                 Want &= ~(1 << (status.MPI_SOURCE - 1));
46:                 break;
47:             case READ_WANT:
48:                 MPI_Send(&Want, 1, MPI_INT, status.MPI_SOURCE,
49:                         READ_WANT, MPI_COMM_WORLD);
50:                 break;
51:             case WRITE_FAST:
52:                 Fast_lock = message;
53:                 break;
54:             case READ_FAST:
55:                 MPI_Send(&Fast_lock, 1, MPI_INT, status.MPI_SOURCE,
56:                         READ_FAST, MPI_COMM_WORLD);
57:                 break;
58:             case WRITE_SLOW:
59:                 Slow_lock = message;
60:                 break;
61:             case READ_SLOW:
62:                 MPI_Send(&Slow_lock, 1, MPI_INT, status.MPI_SOURCE,
63:                         READ_SLOW, MPI_COMM_WORLD);
64:                 break;
65:             case QUIT:
66:                 n -= 1;
67:                 break;
68:         }
69:         if (n == 1) break;
70:     }
71: } else {
72:     /* The participating processors */
73:     for (i = 0; i < 10; i++) {
74:
75:         printf("REMAINDER: %d\n", my_rank - 1);
76:         fflush(stdout);
77:         sleep(1);
78:
79:         sleep(2); /* Do remaining activities */
80:
81:         printf("ENTRY: %d\n", my_rank - 1);
82:         fflush(stdout);
83:         sleep(1);
84:
85:         Start:
86:         MPI_Send(&message, 1, MPI_INT, 0, SET_WANT,
87:                 MPI_COMM_WORLD);
88:         message = my_rank - 1;
89:         MPI_Send(&message, 1, MPI_INT, 0, WRITE_FAST,
90:                 MPI_COMM_WORLD);
91:
92:         MPI_Send(&message, 1, MPI_INT, 0, READ_SLOW,
93:                 MPI_COMM_WORLD);
94:
95:

```

```

96:     MPI_Recv(&message, 1, MPI_INT, 0, READ_SLOW,
97:             MPI_COMM_WORLD, &status);
98:     if (message != -1) {
99:         MPI_Send(&message, 1, MPI_INT, 0, RESET_WANT,
100:                MPI_COMM_WORLD);
101:         do {
102:             MPI_Send(&message, 1, MPI_INT, 0, READ_SLOW,
103:                    MPI_COMM_WORLD);
104:             MPI_Recv(&message, 1, MPI_INT, 0, READ_SLOW,
105:                    MPI_COMM_WORLD, &status);
106:         } while (message != -1);
107:         goto Start;
108:     }
109:
110:     message = my_rank - 1;
111:     MPI_Send(&message, 1, MPI_INT, 0, WRITE_SLOW,
112:            MPI_COMM_WORLD);
113:
114:     MPI_Send(&message, 1, MPI_INT, 0, READ_FAST,
115:            MPI_COMM_WORLD);
116:     MPI_Recv(&message, 1, MPI_INT, 0, READ_FAST,
117:            MPI_COMM_WORLD, &status);
118:     if (message != my_rank - 1) {
119:         MPI_Send(&message, 1, MPI_INT, 0, RESET_WANT,
120:                MPI_COMM_WORLD);
121:         do {
122:             MPI_Send(&message, 1, MPI_INT, 0, READ_WANT,
123:                    MPI_COMM_WORLD);
124:             MPI_Recv(&message, 1, MPI_INT, 0, READ_WANT,
125:                    MPI_COMM_WORLD, &status);
126:         } while (message != 0);
127:         MPI_Send(&message, 1, MPI_INT, 0, READ_SLOW,
128:                MPI_COMM_WORLD);
129:         MPI_Recv(&message, 1, MPI_INT, 0, READ_SLOW,
130:                MPI_COMM_WORLD, &status);
131:         if (message != my_rank - 1) {
132:             do {
133:                 MPI_Send(&message, 1, MPI_INT, 0, READ_SLOW,
134:                        MPI_COMM_WORLD);
135:                 MPI_Recv(&message, 1, MPI_INT, 0, READ_SLOW,
136:                        MPI_COMM_WORLD, &status);
137:             } while (message != -1);
138:             goto Start;
139:         }
140:     }
141:
142:     printf("CRITICAL: %d\n", my_rank - 1);
143:     fflush(stdout);
144:     sleep(1);
145:
146:     sleep(2); /* Do critical section */
147:
148:     printf("EXIT: %d\n", my_rank - 1);
149:     fflush(stdout);
150:     sleep(1);
151:
152:     message = -1;
153:     MPI_Send(&message, 1, MPI_INT, 0, WRITE_SLOW,
154:            MPI_COMM_WORLD);

```

```

155:     MPI_Send(&message, 1, MPI_INT, 0, RESET_WANT,
156:             MPI_COMM_WORLD);
157: }
158:
159:     MPI_Send(&message, 1, MPI_INT, 0, QUIT,
160:             MPI_COMM_WORLD);
161: }
162:
163: MPI_Finalize();
164: } /* main */
165:

```

### 3.2 Alur and Taubenfeld's Algorithm

Alur and Taubenfeld's algorithm uses assumption on upper bound of message transmission. We assume this upper bound to be 5 seconds, which is long enough for most cases. There is inherent danger in the algorithm that at some time of the day the delay could be longer than 5 seconds due to various reasons, however, this assumption is permissible if the algorithm is executed by embedded system, or an operating system with hardware interrupts disabled [11].

```

1: /*
2:    Alur and Taubenfeld's Fast
3:    Mutual Exclusion Algorithm
4:    */
5:
6: #include <stdio.h>
7: #include "mpi.h"
8:
9: /* Message tags */
10: #define READ_FAST 50
11: #define WRITE_FAST 51
12: #define READ_SLOW 52
13: #define WRITE_SLOW 53
14: #define READ_Z 54
15: #define SET_Z 55
16: #define RESET_Z 56
17:
18: #define QUIT 57
19:
20: main(int argc, char** argv) {
21:     int my_rank; /* Rank of processor */
22:     int n; /* No. of processors */
23:     int i;
24:     int message; /* Storage for the message */
25:     MPI_Status status; /* Return status for the receive */
26:
27:     MPI_Init(&argc, &argv);
28:     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
29:     MPI_Comm_size(MPI_COMM_WORLD, &n);
30:
31:     if (!my_rank) {
32:         int Fast_lock = -1;
33:         int Slow_lock = -1;
34:         int Z = 0;

```

```

35:
36:   for (;;) {
37:     MPI_Recv(&message, 1, MPI_INT, MPI_ANY_SOURCE,
38:             MPI_ANY_TAG, MPI_COMM_WORLD, &status);
39:     switch (status.MPI_TAG) {
40:
41:     case WRITE_FAST:
42:       Fast_lock = message;
43:       break;
44:     case READ_FAST:
45:       MPI_Send(&Fast_lock, 1, MPI_INT, status.MPI_SOURCE,
46:              READ_FAST, MPI_COMM_WORLD);
47:       break;
48:     case WRITE_SLOW:
49:       Slow_lock = message;
50:       break;
51:     case READ_SLOW:
52:       MPI_Send(&Slow_lock, 1, MPI_INT, status.MPI_SOURCE,
53:              READ_SLOW, MPI_COMM_WORLD);
54:       break;
55:     case SET_Z:
56:       Z = -1;
57:       break;
58:     case RESET_Z:
59:       Z = 0;
60:       break;
61:     case READ_Z:
62:       MPI_Send(&Z, 1, MPI_INT, status.MPI_SOURCE,
63:              READ_Z, MPI_COMM_WORLD);
64:       break;
65:     case QUIT:
66:       n -= 1;
67:       break;
68:     }
69:     if (n == 1) break;
70:   }
71: } else {
72:   for (i = 0; i < 10; i++) {
73:
74:     printf("REMAINDER: %d\n", my_rank - 1);
75:     sleep(1);
76:     fflush(stdout);
77:
78:     sleep(2); /* Do remaining activities */
79:
80:     printf("ENTRY: %d\n", my_rank - 1);
81:     sleep(1);
82:     fflush(stdout);
83:
84:   Start:
85:     message = my_rank - 1;
86:     MPI_Send(&message, 1, MPI_INT, 0, WRITE_FAST,
87:            MPI_COMM_WORLD);
88:
89:     do {
90:       MPI_Send(&message, 1, MPI_INT, 0, READ_SLOW,
91:              MPI_COMM_WORLD);
92:
93:

```

```

94:     MPI_Recv(&message, 1, MPI_INT, 0, READ_SLOW,
95:             MPI_COMM_WORLD, &status);
96: } while (message != -1);
97:
98: message = my_rank - 1;
99: MPI_Send(&message, 1, MPI_INT, 0, WRITE_SLOW,
100:         MPI_COMM_WORLD);
101:
102: MPI_Send(&message, 1, MPI_INT, 0, READ_FAST,
103:         MPI_COMM_WORLD);
104: MPI_Recv(&message, 1, MPI_INT, 0, READ_FAST,
105:         MPI_COMM_WORLD, &status);
106: if (message != my_rank - 1) {
107:
108:     sleep(5); /* 5 seconds delay should be enough */
109:
110:     MPI_Send(&message, 1, MPI_INT, 0, READ_SLOW,
111:            MPI_COMM_WORLD);
112:     MPI_Recv(&message, 1, MPI_INT, 0, READ_SLOW,
113:            MPI_COMM_WORLD, &status);
114:     if (message != my_rank - 1) goto Start;
115:
116:     do {
117:         MPI_Send(&message, 1, MPI_INT, 0, READ_Z,
118:                MPI_COMM_WORLD);
119:         MPI_Recv(&message, 1, MPI_INT, 0, READ_Z,
120:                MPI_COMM_WORLD, &status);
121:     } while (message);
122:
123: } else {
124:     MPI_Send(&message, 1, MPI_INT, 0, SET_Z,
125:            MPI_COMM_WORLD);
126: }
127:
128: printf("CRITICAL: %d\n", my_rank - 1);
129: sleep(1);
130: fflush(stdout);
131:
132: sleep(2); /* Do critical section */
133:
134: printf("EXIT: %d\n", my_rank - 1);
135: sleep(1);
136: fflush(stdout);
137:
138: MPI_Send(&message, 1, MPI_INT, 0, RESET_Z,
139:         MPI_COMM_WORLD);
140:
141: MPI_Send(&message, 1, MPI_INT, 0, READ_SLOW,
142:         MPI_COMM_WORLD);
143: MPI_Recv(&message, 1, MPI_INT, 0, READ_SLOW,
144:         MPI_COMM_WORLD, &status);
145: if (message == my_rank - 1) {
146:     message = -1;
147:     MPI_Send(&message, 1, MPI_INT, 0, WRITE_SLOW,
148:            MPI_COMM_WORLD);
149: }
150: }
151:
152: MPI_Send(&message, 1, MPI_INT, 0, QUIT,

```

```

153:             MPI_COMM_WORLD);
154:     }
155:
156:     MPI_Finalize();
157: } /* main */
158:

```

### 3.3 Michael and Scott's Algorithm

Similar to the implementation of Alur and Taubenfeld's algorithm, here we use 5 seconds delay in line 122 of the following code.

```

1: /*
2:    Michael and Scott's
3:    Fast Mutual Exclusion Algorithm
4:    */
5:
6: #include <stdio.h>
7: #include "mpi.h"
8:
9: /* Message tags */
10: #define READ_FAST 50
11: #define WRITE_FAST 51
12: #define READ_SLOW 52
13: #define WRITE_SLOW 53
14: #define READ_F 54
15: #define WRITE_F 55
16: #define READ_SLOW_F 56
17: #define WRITE_SLOW_F 57
18:
19: #define QUIT 58
20:
21: #define IN 0;
22: #define OUT -1;
23:
24: main(int argc, char** argv) {
25:     int my_rank; /* Rank of processor */
26:     int n; /* No. of processors */
27:     int i;
28:     int message[2]; /* Storage for the message */
29:     MPI_Status status; /* Return status for the receive */
30:
31:     MPI_Init(&argc, &argv);
32:     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
33:     MPI_Comm_size(MPI_COMM_WORLD, &n);
34:
35:     if (!my_rank) {
36:         int Fast_lock = -1;
37:         int Slow_lock = -1;
38:         int F = OUT;
39:
40:         for (;;) {
41:             MPI_Recv(message, 2, MPI_INT, MPI_ANY_SOURCE,
42:                     MPI_ANY_TAG, MPI_COMM_WORLD, &status);
43:             switch (status.MPI_TAG) {
44:
45:                 case WRITE_FAST:

```

```

46:     Fast_lock = *message;
47:     break;
48: case READ_FAST:
49:     *message = Fast_lock;
50:     MPI_Send(message, 2, MPI_INT, status.MPI_SOURCE,
51:             READ_FAST, MPI_COMM_WORLD);
52:     break;
53: case WRITE_SLOW:
54:     Slow_lock = *message;
55:     break;
56: case READ_SLOW:
57:     *message = Slow_lock;
58:     MPI_Send(message, 2, MPI_INT, status.MPI_SOURCE,
59:             READ_SLOW, MPI_COMM_WORLD);
60:     break;
61: case WRITE_F:
62:     F = *message;
63:     break;
64: case READ_F:
65:     *message = F;
66:     MPI_Send(message, 2, MPI_INT, status.MPI_SOURCE,
67:             READ_F, MPI_COMM_WORLD);
68:     break;
69: case WRITE_SLOW_F:
70:     Slow_lock = *message;
71:     F = *(message + 1);
72:     break;
73: case READ_SLOW_F:
74:     *message = Slow_lock;
75:     *(message + 1) = F;
76:     MPI_Send(message, 2, MPI_INT, status.MPI_SOURCE,
77:             READ_SLOW_F, MPI_COMM_WORLD);
78:     break;
79: case QUIT:
80:     n -= 1;
81:     break;
82: }
83: if (n == 1) break;
84: }
85:
86: } else {
87:
88:     for (i = 0; i < 10; i++) {
89:
90:         printf("REMAINDER: %d\n", my_rank - 1);
91:         sleep(1);
92:         fflush(stdout);
93:
94:         sleep(2); /* Do remaining activities */
95:
96:         printf("ENTRY: %d\n", my_rank - 1);
97:         sleep(1);
98:         fflush(stdout);
99:
100:     Start:
101:         *message = my_rank - 1;
102:         MPI_Send(message, 2, MPI_INT, 0, WRITE_FAST,
103:                 MPI_COMM_WORLD);
104:

```

```

105:     do {
106:         MPI_Send(message, 2, MPI_INT, 0, READ_SLOW,
107:                 MPI_COMM_WORLD);
108:         MPI_Recv(message, 2, MPI_INT, 0, READ_SLOW,
109:                 MPI_COMM_WORLD, &status);
110:     } while (*message != -1);
111:
112:     *message = my_rank - 1;
113:     MPI_Send(message, 2, MPI_INT, 0, WRITE_SLOW,
114:             MPI_COMM_WORLD);
115:
116:     MPI_Send(message, 2, MPI_INT, 0, READ_FAST,
117:             MPI_COMM_WORLD);
118:     MPI_Recv(message, 2, MPI_INT, 0, READ_FAST,
119:             MPI_COMM_WORLD, &status);
120:     if (*message != my_rank - 1) {
121:
122:         sleep(5); /* 5 seconds delay should be enough */
123:
124:         MPI_Send(message, 2, MPI_INT, 0, READ_SLOW_F,
125:                 MPI_COMM_WORLD);
126:         MPI_Recv(message, 2, MPI_INT, 0, READ_SLOW_F,
127:                 MPI_COMM_WORLD, &status);
128:         if (*message != my_rank - 1) goto Start;
129:     }
130:     *message = IN;
131:     MPI_Send(message, 2, MPI_INT, 0, WRITE_F,
132:             MPI_COMM_WORLD);
133:
134:     printf("CRITICAL: %d\n", my_rank - 1);
135:     sleep(1);
136:     fflush(stdout);
137:
138:     sleep(2); /* Do critical section */
139:
140:     printf("EXIT: %d\n", my_rank - 1);
141:     sleep(1);
142:     fflush(stdout);
143:
144:     *message = -1;
145:     *(message + 1) = OUT;
146:     MPI_Send(message, 2, MPI_INT, 0, WRITE_SLOW_F,
147:             MPI_COMM_WORLD);
148: }
149:
150:     MPI_Send(message, 2, MPI_INT, 0, QUIT,
151:             MPI_COMM_WORLD);
152: }
153:
154:     MPI_Finalize();
155: } /* main */
156:

```

### 3.4 Styer's Algorithm

The implementation is shown below.

```
1: /*
```

```

2:   Styer's Fast Mutual Exclusion Algorithm
3:   */
4:
5: #include <stdio.h>
6: #include <math.h>
7: #include <unistd.h>
8: #include "mpi.h"
9:
10: #define MAX_LEVEL 3
11: #define K 2
12:
13: #define LOG_RANK(R) (R - 1)
14: #define PHYS_RANK(R) (R + 1)
15:
16: #define OUT 0
17: #define FAST 1
18: #define SLOW 2
19:
20: /* Message tags */
21: #define SET_FAST 50
22: #define SET_SLOW 51
23: #define SET_OUT 52
24: #define READ_W 65
25:
26: #define SET_S 53
27:
28: #define SET_TURN 63
29: #define READ_TURN 64
30:
31: #define SET_LOCK 54
32: #define RESET_LOCK 55
33: #define READ_LOCK 56
34:
35: #define SET_BLOCK 57
36: #define RESET_BLOCK 58
37: #define READ_BLOCK 59
38:
39: #define TEST_AND_SET 60
40: #define RESET 61
41:
42: #define QUIT 62
43:
44:
45: /* Array index of a tree node #n of level l */
46: int index(int l, int n)
47: {
48:     return (int)pow(K, (MAX_LEVEL - l + 1)) - 2 - n;
49: }
50:
51: /* The entry part in a test&set mutual exclusion. */
52: void ts_entry(int l, int n, int vid)
53:     /* l = level of tree,
54:        n = node # in level l,
55:        vid = virtual id, not used since we don't
56:        care about processor id in test&set mutual
57:        exclusion. */
58: {
59:     int sendbuf[2], recvbuf;
60:     MPI_Status status;

```

```

61:
62:   do {
63:     *sendbuf = 1;
64:     *(sendbuf + 1) = n;
65:     MPI_Sendrecv(sendbuf, 2, MPI_INT, 0, TEST_AND_SET,
66:                 &recvbuf, 1, MPI_INT, 0, TEST_AND_SET,
67:                 MPI_COMM_WORLD, &status);
68:   } while (recvbuf);
69: }
70:
71: /* The exit part in a testset mutual exclusion. */
72: void ts_exit(int l, int n, int vid)
73:     /* l = level of tree,
74:        n = node # in level l,
75:        vid = virtual id, not used since we don't
76:        care about processor id in test&set mutual
77:        exclusion. */
78: {
79:   int message[2];
80:
81:   *message = 1;
82:   *(message + 1) = n;
83:   MPI_Send(message, 2, MPI_INT, 0, RESET, MPI_COMM_WORLD);
84: }
85:
86: void check_vars(int l, int start)
87: {
88:   int j;
89:   int sendbuf[2], recvbuf;
90:   MPI_Status status;
91:
92:   if (l > 0)
93:     for (j = 0; j < K; j++) {
94:       do {
95:         *sendbuf = 1;
96:         *(sendbuf + 1) = start + j;
97:         MPI_Sendrecv(sendbuf, 2, MPI_INT, 0, READ_W,
98:                     &recvbuf, 1, MPI_INT, 0, READ_W,
99:                     MPI_COMM_WORLD, &status);
100:       } while (recvbuf == FAST);
101:       if (recvbuf == SLOW)
102:         check_vars(l - 1, (start + j) * K);
103:     }
104:   else
105:     for (j = 0; j < K; j++)
106:       do {
107:         *sendbuf = 1;
108:         *(sendbuf + 1) = start + j;
109:         MPI_Sendrecv(sendbuf, 2, MPI_INT, 0, READ_W,
110:                     &recvbuf, 1, MPI_INT, 0, READ_W,
111:                     MPI_COMM_WORLD, &status);
112:       } while (recvbuf == FAST);
113: }
114:
115: main(int argc, char** argv) {
116:   int my_rank; /* Rank of processor */
117:   int n; /* No. of processors */
118:   int i;
119:   int j;

```

```

120: int message[2]; /* Storage for the message */
121: int reply;
122: MPI_Status status; /* Return status for the receive */
123:
124: MPI_Init(&argc, &argv);
125: MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
126: MPI_Comm_size(MPI_COMM_WORLD, &n);
127:
128: if (!my_rank) {
129:     int *V; /* The test&set registers */
130:     int *S;
131:     int *W;
132:     int Lock = 0;
133:     int Block = 0;
134:     int Turn = -1;
135:
136:     V = (int *)malloc(sizeof(int) * ((int)pow(K, (MAX_LEVEL + 1)) - 1));
137:     S = (int *)malloc(sizeof(int) * ((int)pow(K, (MAX_LEVEL + 1)) - 1));
138:
139:     /* All test&set registers are 0, and S are all nil. */
140:     for (i = 0; i < (int)pow(K, MAX_LEVEL + 1) - 1; i++) {
141:         V[i] = 0;
142:         S[i] = -1;
143:     }
144:
145:     W = (int *)malloc(sizeof(int) * n);
146:
147:     /* Initially, all processors are in the remainder section */
148:     for (i = 0; i < n; i++) W[i] = OUT;
149:
150:     for (;;) {
151:         MPI_Recv(message, 2, MPI_INT, MPI_ANY_SOURCE,
152:                 MPI_ANY_TAG, MPI_COMM_WORLD, &status);
153:         switch (status.MPI_TAG) {
154:             case TEST_AND_SET:
155:                 {
156:                     int temp = *(V + index(*message, *(message + 1)));
157:                     *(V + index(*message, *(message + 1))) = 1;
158:                     MPI_Send(&temp, 1, MPI_INT, status.MPI_SOURCE, TEST_AND_SET, MPI_COMM_WORLD);
159:                 }
160:                 break;
161:             case RESET:
162:                 *(V + index(*message, *(message + 1))) = 0;
163:                 break;
164:             case SET_TURN:
165:                 Turn = LOG_RANK(status.MPI_SOURCE);
166:                 break;
167:             case READ_TURN:
168:                 reply = Turn;
169:                 MPI_Send(&reply, 1, MPI_INT, status.MPI_SOURCE, READ_TURN, MPI_COMM_WORLD);
170:                 break;
171:             case SET_FAST:
172:                 *(W + LOG_RANK(status.MPI_SOURCE)) = FAST;
173:                 break;
174:             case SET_SLOW:
175:                 *(W + LOG_RANK(status.MPI_SOURCE)) = SLOW;
176:                 break;
177:             case SET_OUT:
178:                 *(W + LOG_RANK(status.MPI_SOURCE)) = OUT;

```

```

179:         break;
180:     case READ_W:
181:         reply = *(W + *(S + index(*message, *(message + 1))));
182:         MPI_Send(&reply, 1, MPI_INT, status.MPI_SOURCE, READ_W, MPI_COMM_WORLD);
183:         break;
184:     case SET_S:
185:         *(S + index(*message, *(message + 1))) = LOG_RANK(status.MPI_SOURCE);
186:         break;
187:     case SET_LOCK:
188:         Lock = 1;
189:         break;
190:     case RESET_LOCK:
191:         Lock = 0;
192:         break;
193:     case READ_LOCK:
194:         reply = Lock;
195:         MPI_Send(&reply, 1, MPI_INT, status.MPI_SOURCE, READ_LOCK, MPI_COMM_WORLD);
196:         break;
197:     case SET_BLOCK:
198:         Block = 1;
199:         break;
200:     case RESET_BLOCK:
201:         Block = 0;
202:         break;
203:     case READ_BLOCK:
204:         reply = Block;
205:         MPI_Send(&reply, 1, MPI_INT, status.MPI_SOURCE, READ_LOCK, MPI_COMM_WORLD);
206:         break;
207:     case QUIT:
208:         n -= 1;
209:         break;
210:     }
211:     if (n == 1) break;
212: }
213:
214: free(V); /* Free allocated memory */
215: free(S);
216: free(W);
217:
218: } else {
219:
220:     /* The participating processors */
221:
222:     for (i = 0; i < 10; i++) {
223:
224:         printf("REMAINDER: %d\n", my_rank - 1);
225:         fflush(stdout);
226:         sleep(1);
227:
228:         sleep(2); /* Do remaining activities */
229:
230:         printf("ENTRY: %d\n", my_rank - 1);
231:         fflush(stdout);
232:         sleep(1);
233:
234:         MPI_Send(NULL, 0, MPI_INT, 0, SET_FAST, MPI_COMM_WORLD);
235:         MPI_Send(NULL, 0, MPI_INT, 0, SET_TURN, MPI_COMM_WORLD);
236:         MPI_Sendrecv(NULL, 0, MPI_INT, 0, READ_LOCK,
237:                     &reply, 1, MPI_INT, 0, READ_LOCK,

```

```

238:             MPI_COMM_WORLD, &status);
239:
240:     if (*message) goto Aside;
241:     for (j = 1; j < MAX_LEVEL; j++) {
242:         *message = j;
243:         *(message + 1) = (int)floor(i / (int)pow(K, j));
244:         MPI_Send(message, 2, MPI_INT, 0, SET_S, MPI_COMM_WORLD);
245:     }
246:
247:     MPI_Send(NULL, 0, MPI_INT, 0, SET_LOCK, MPI_COMM_WORLD);
248:
249:     MPI_Sendrecv(NULL, 0, MPI_INT, 0, READ_TURN,
250:                 &reply, 1, MPI_INT, 0, READ_TURN,
251:                 MPI_COMM_WORLD, &status);
252:     if (reply != LOG_RANK(my_rank)) goto Aside;
253:
254:     MPI_Sendrecv(NULL, 0, MPI_INT, 0, READ_BLOCK,
255:                 &reply, 1, MPI_INT, 0, READ_BLOCK,
256:                 MPI_COMM_WORLD, &status);
257:     if (reply) goto Aside;
258:
259:
260:     printf("CRITICAL: %d\n", my_rank - 1);
261:     fflush(stdout);
262:     sleep(1);
263:
264:     sleep(2); /* Do critical section */
265:
266:     printf("EXIT: %d\n", my_rank - 1);
267:     fflush(stdout);
268:     sleep(1);
269:
270:     MPI_Send(NULL, 0, MPI_INT, 0, RESET_LOCK, MPI_COMM_WORLD);
271:     MPI_Send(NULL, 0, MPI_INT, 0, SET_OUT, MPI_COMM_WORLD);
272:
273:     continue;
274:
275: Aside:
276:     MPI_Send(NULL, 0, MPI_INT, 0, SET_SLOW, MPI_COMM_WORLD);
277:     for (j = 0; j < MAX_LEVEL; j++) {
278:         ts_entry(j, (int)floor(LOG_RANK(my_rank) / (int)pow(K, j + 1)),
279:                 (int)floor(LOG_RANK(my_rank) / (int)pow(K, j)));
280:     }
281:
282:     MPI_Send(NULL, 0, MPI_INT, 0, SET_BLOCK, MPI_COMM_WORLD);
283:     check_vars(MAX_LEVEL - 1, 0);
284:
285:     printf("CRITICAL: %d\n", my_rank - 1);
286:     fflush(stdout);
287:     sleep(1);
288:
289:     sleep(2); /* Do critical section */
290:
291:     printf("EXIT: %d\n", my_rank - 1);
292:     fflush(stdout);
293:     sleep(1);
294:
295:     MPI_Send(NULL, 0, MPI_INT, 0, RESET_LOCK, MPI_COMM_WORLD);
296:     MPI_Send(NULL, 0, MPI_INT, 0, RESET_BLOCK, MPI_COMM_WORLD);

```

```

297:     MPI_Send(NULL, 0, MPI_INT, 0, SET_OUT, MPI_COMM_WORLD);
298:     for (j = MAX_LEVEL - 1; j >= 0; j--) {
299:         ts_exit(j, (int)floor(LOG_RANK(my_rank) / (int)pow(K, j + 1)),
300:             (int)floor(LOG_RANK(my_rank) / (int)pow(K, j)));
301:     }
302:
303:     continue;
304: }
305:
306: MPI_Send(message, 2, MPI_INT, 0, QUIT, MPI_COMM_WORLD);
307: }
308:
309: MPI_Finalize();
310: } /* main */
311:

```

### 3.5 MCS Lock

The locally-accessible shared memory are very naturally implemented using MPI's blocking receive *MPI\_Recv*. We replace the whole spin of lines 6 and 10 of Figure 8 with the blocking receives of lines 124 and 150.

The shared memory server only serves memory read/write through *fetch\_and\_store* and *compare\_and\_swap* functions that implements atomic operations *fetch&store* and *compare&swap*, respectively.

In each processor, we do not actually using *qnode*. Instead, we replace it with simply its element variable *next*. We do not need to implement the variable *locked* since it is only useful for signaling a spinning processor (in which case we can use MPI's blocked send and receive pair in lines 155 and 124).

```

1: /*
2:    Mellor-Crummey and Scott's Lock
3:    */
4:
5: #include <stdio.h>
6: #include "mpi.h"
7:
8: #define PHYS_RANK(R) R + 1
9: #define LOG_RANK(R) R - 1
10:
11: /* Message tags */
12: #define FETCH_AND_STORE 50
13: #define COMPARE_AND_SWAP 51
14:
15: #define WRITE_NEXT 52
16: #define RESET_LOCKED 53
17:
18: #define QUIT 54
19:
20: /* Fetch&store atomic operation */
21: int fetch_and_store(int rank)
22: {
23:     int message[2];
24:     MPI_Status status;
25:

```

```

26:  *message = rank;
27:  MPI_Send(message, 2, MPI_INT, 0,
28:          FETCH_AND_STORE, MPI_COMM_WORLD);
29:  MPI_Recv(message, 2, MPI_INT, 0,
30:          FETCH_AND_STORE, MPI_COMM_WORLD, &status);
31:  return *message;
32: }
33:
34: /* Compare&swap atomic operation */
35: int compare_and_swap(int rank1, int rank2)
36: {
37:     int message[2];
38:     MPI_Status status;
39:
40:     *message = rank1;
41:     *(message + 1) = rank2;
42:     MPI_Send(message, 2, MPI_INT, 0,
43:             COMPARE_AND_SWAP, MPI_COMM_WORLD);
44:     MPI_Recv(message, 2, MPI_INT, 0,
45:             COMPARE_AND_SWAP, MPI_COMM_WORLD, &status);
46:     return *message;
47: }
48:
49:
50: main(int argc, char** argv) {
51:     int my_rank; /* Rank of processor */
52:     int n; /* No. of processors */
53:     int i;
54:     int message[2]; /* Storage for the message */
55:     int flag;
56:     MPI_Status status; /* Return status for the receive */
57:     MPI_Request request;
58:
59:     MPI_Init(&argc, &argv);
60:     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
61:     MPI_Comm_size(MPI_COMM_WORLD, &n);
62:
63:
64:     if (!my_rank) {
65:         /* The shared memory server */
66:
67:         int Lock = -1;
68:         int tmp;
69:
70:         for (;;) {
71:             MPI_Recv(message, 2, MPI_INT, MPI_ANY_SOURCE,
72:                     MPI_ANY_TAG, MPI_COMM_WORLD, &status);
73:             switch (status.MPI_TAG) {
74:
75:                 case FETCH_AND_STORE: /* Service fetch&store */
76:                     tmp = Lock;
77:                     Lock = *message;
78:                     *message = tmp;
79:                     MPI_Send(message, 2, MPI_INT, status.MPI_SOURCE,
80:                             FETCH_AND_STORE, MPI_COMM_WORLD);
81:                     break;
82:                 case COMPARE_AND_SWAP: /* Service compare&swap */
83:                     tmp = Lock;
84:                     if (Lock == *message)

```

```

85:         Lock = *(message + 1);
86:         *message = tmp;
87:         MPI_Send(message, 2, MPI_INT, status.MPI_SOURCE,
88:                 COMPARE_AND_SWAP, MPI_COMM_WORLD);
89:         break;
90:     case QUIT:
91:         n -= 1;
92:         break;
93:     }
94:     if (n == 1) break;
95: }
96:
97: } else {
98:     int next = -1;
99:     int predecessor = -1;
100:
101:     for (i = 0; i < 10; i++) {
102:
103:         printf("REMAINDER: %d\n", LOG_RANK(my_rank));
104:         fflush(stdout);
105:         sleep(1);
106:
107:         sleep(2); /* Do remaining activities */
108:
109:         printf("ENTRY: %d\n", LOG_RANK(my_rank));
110:         fflush(stdout);
111:         sleep(1);
112:
113:         next = -1;
114:         predecessor = fetch_and_store(LOG_RANK(my_rank));
115:         if (predecessor != -1) {
116:             /* There is a predecessor */
117:
118:             /* Set myself as predecessor's next */
119:             *message = LOG_RANK(my_rank);
120:             MPI_Send(message, 2, MPI_INT, PHYS_RANK(predecessor),
121:                     WRITE_NEXT, MPI_COMM_WORLD);
122:
123:             /* Wait until unlocked */
124:             MPI_Recv(message, 2, MPI_INT, MPI_ANY_SOURCE,
125:                     RESET_LOCKED, MPI_COMM_WORLD, &status);
126:         }
127:
128:         printf("CRITICAL: %d\n", LOG_RANK(my_rank));
129:         fflush(stdout);
130:         sleep(1);
131:
132:         sleep(2); /* Do critical section */
133:
134:         printf("EXIT: %d\n", LOG_RANK(my_rank));
135:         fflush(stdout);
136:         sleep(1);
137:
138:         /* So that we do not have to always enter the if loop */
139:         MPI_Irecv(message, 2, MPI_INT, MPI_ANY_SOURCE,
140:                  WRITE_NEXT, MPI_COMM_WORLD, &request);
141:         MPI_Test(&request, &flag, &status);
142:         if (flag) {
143:             next = *message;

```

```

144:     }
145:
146:     if (next == -1) {
147:         if (compare_and_swap(LOG_RANK(my_rank), -1) == -1)
148:             continue; /* Goto Remainder */
149:
150:         MPI_Recv(message, 2, MPI_INT, MPI_ANY_SOURCE,
151:                 WRITE_NEXT, MPI_COMM_WORLD, &status);
152:         next = *message;
153:     }
154:
155:     MPI_Send(message, 2, MPI_INT, PHYS_RANK(next),
156:             RESET_LOCKED, MPI_COMM_WORLD);
157: }
158:
159: MPI_Send(message, 2, MPI_INT, 0, QUIT, MPI_COMM_WORLD);
160: }
161:
162: MPI_Finalize();
163: }

```

### 3.6 Huang's Algorithm

According to the paper [6], incarnation bit is the least-significant bit (LSB) of a data address. In our implementation, we used most-significant bit (MSB) of an integer instead since in the C programming language, programmers have no control over LSB of data addresses.

Similar to the implementation of MCS lock, we do not actually implement the permission word data type as well as pointers. For the contents of the permission word, we simply use two variables `head` and `tail`, and the pointer is simply the rank of the processor with MSB set according to incarnation bit.

```

1: /*
2:    Huang's Algorithm for Mutual Exclusion
3:    */
4:
5: #include <stdio.h>
6: #include "mpi.h"
7:
8: /* Message tags */
9: #define FETCH_AND_STORE 50
10: #define COMPARE_AND_SWAP 51
11:
12: #define SET_PWORD 52
13:
14: #define QUIT 53
15:
16:
17: /* Physical rank of a logical rank r,
18:    possibly with incarnation bit set. */
19: int phys_rank(int r) {
20:     return (r & ~(1 << (sizeof(int) * 8 - 1))) + 1;
21: }
22:
23: /* Logical rank of a physical rank r,

```

```

24:     set incarnation bit when necessary. */
25: int log_rank(int r, int bit) {
26:     if (bit) {
27:         return (r - 1) | (1 << (sizeof(int) * 8 - 1));
28:     }
29:     return r - 1;
30: }
31:
32: /* Fetch&store atomic operation to be
33:    executed by clients */
34: int fetch_and_store(int rank)
35: {
36:     int message[2];
37:     MPI_Status status;
38:
39:     *message = rank;
40:     MPI_Send(message, 2, MPI_INT, 0,
41:              FETCH_AND_STORE, MPI_COMM_WORLD);
42:     MPI_Recv(message, 2, MPI_INT, 0,
43:              FETCH_AND_STORE, MPI_COMM_WORLD, &status);
44:     return *message;
45: }
46:
47: /* Compare&swap atomic operation to be
48:    executed by clients */
49: int compare_and_swap(int rank1, int rank2)
50: {
51:     int message[2];
52:     MPI_Status status;
53:
54:     *message = rank1;
55:     *(message + 1) = rank2;
56:     MPI_Send(message, 2, MPI_INT, 0,
57:              COMPARE_AND_SWAP, MPI_COMM_WORLD);
58:     MPI_Recv(message, 2, MPI_INT, 0,
59:              COMPARE_AND_SWAP, MPI_COMM_WORLD, &status);
60:     return *message;
61: }
62:
63:
64: main(int argc, char** argv) {
65:     int my_rank; /* Rank of processor */
66:     int n; /* No. of processors */
67:     int i;
68:     int message[2]; /* Storage for the message */
69:     MPI_Status status; /* Return status for the receive */
70:     MPI_Request request;
71:
72:     MPI_Init(&argc, &argv);
73:     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
74:     MPI_Comm_size(MPI_COMM_WORLD, &n);
75:
76:     if (!my_rank) {
77:         /* The shared memory server */
78:
79:         int Lock = -1;
80:         int tmp;
81:
82:         for (;;) {

```

```

83:     MPI_Recv(message, 2, MPI_INT, MPI_ANY_SOURCE,
84:             MPI_ANY_TAG, MPI_COMM_WORLD, &status);
85:     switch (status.MPI_TAG) {
86:
87:     case FETCH_AND_STORE:
88:         tmp = Lock;
89:         Lock = *message;
90:         *message = tmp;
91:         MPI_Send(message, 2, MPI_INT, status.MPI_SOURCE,
92:                 FETCH_AND_STORE, MPI_COMM_WORLD);
93:         break;
94:     case COMPARE_AND_SWAP:
95:         tmp = Lock;
96:         if (Lock == *message)
97:             Lock = *(message + 1);
98:         *message = tmp;
99:         MPI_Send(message, 2, MPI_INT, status.MPI_SOURCE,
100:                 COMPARE_AND_SWAP, MPI_COMM_WORLD);
101:         break;
102:     case QUIT:
103:         n -= 1;
104:         break;
105:     }
106:     if (n == 1) break;
107: }
108:
109: } else {
110:     /* The clients */
111:
112:     int myaddr; /* The logical address of this processor,
113:                i.e., the address of this processor's
114:                permission word, possibly with
115:                incarnation bit set. */
116:     int incarnation = 0; /* The incarnation bit, */
117:                        /* initially 0. */
118:     int head = -1; /* The contents of permission */
119:     int tail = -1; /* word (head,tail). */
120:     int next = -1; /* The address of next processor's */
121:                  /* permission bit */
122:
123:     for (i = 0; i < 10; i++) {
124:
125:         printf("REMAINDER: %d\n", my_rank - 1);
126:         fflush(stdout);
127:         sleep(1);
128:
129:         sleep(2); /* Do remaining activities */
130:
131:         printf("ENTRY: %d\n", my_rank - 1);
132:         fflush(stdout);
133:         sleep(1);
134:
135:         /* Inverse incarnation */
136:         incarnation = -(incarnation - 1);
137:
138:         /* Logical rank with incarnation */
139:         myaddr = log_rank(my_rank, incarnation);
140:
141:         /* Initially, permission word is nil */

```

```

142:     head = tail = -1;
143:
144:     next = fetch_and_store(myaddr);
145:     if (next != -1) {
146:         MPI_Recv(message, 2, MPI_INT, MPI_ANY_SOURCE,
147:                 SET_PWORD, MPI_COMM_WORLD, &status);
148:         head = *message;
149:         tail = *(message + 1);
150:     }
151:
152:     printf("CRITICAL: %d\n", my_rank - 1);
153:     fflush(stdout);
154:     sleep(1);
155:
156:     sleep(2); /* Do critical section */
157:
158:     printf("EXIT: %d\n", my_rank - 1);
159:     fflush(stdout);
160:     sleep(1);
161:
162:     if (next == -1) {
163:     CompSwap:
164:         next = compare_and_swap(myaddr, -1);
165:         if (next != myaddr) {
166:             /* Generate permission word and pass
167:              to next processor */
168:             *message = myaddr;
169:             *(message + 1) = next;
170:             MPI_Send(message, 2, MPI_INT, phys_rank(next),
171:                     SET_PWORD, MPI_COMM_WORLD);
172:         }
173:     } else {
174:         if (next == head) {
175:             myaddr = tail;
176:             goto CompSwap;
177:         }
178:
179:         /* Pass permission word to next processor */
180:         *message = head;
181:         *(message + 1) = tail;
182:         MPI_Send(message, 2, MPI_INT, phys_rank(next),
183:                 SET_PWORD, MPI_COMM_WORLD);
184:     }
185: }
186: MPI_Send(message, 2, MPI_INT, 0, QUIT,
187:         MPI_COMM_WORLD);
188: }
189:
190: MPI_Finalize();
191: }

```

## 4 Comparisons

Comparisons table of the algorithm that we present is shown in Figure 12. The table is created based on the trace output of the actual execution of each algo-

Algorithm Name	Deadlock Detected	Unfairness Detected	Access at No Contention	Access at Contention
Lamport's	No	Yes	5	$O(n)$
Alur and Taubenfeld's	No	Yes	5	$O(n)$
Michael and Scott's	No	Yes	5	$O(n)$
MCS Lock	No	Yes	1	2
Huang's	No	No	1	1

Figure 12: Comparisons Table

rithm. Unfairness was detected at execution of Lamport's, Alur and Taubenfeld's, and Michael and Scott's algorithms since some processors are allowed to execute the critical section until it finishes iterating before other processors may enter the critical section. The algorithms therefore do not guarantee fairness.

On the other hand, MCS lock guarantees FIFO ordering, as can also be observed from the actual execution trace. However, the program did not terminate due to the last processor waiting for another process to signal it to leave its exit section, as explained in Section 3.5.

Although some literature suggests that Lamport's algorithm has  $O(n^2)$  memory access time during contention [11]. Due to the fact that we use integer bits instead of array to implement the *Want* shared variable, we were able to reduce the memory access to  $O(n)$ .

In MCS lock and Huang's algorithm, only one fetch&store is needed when there is no contention to be able to enter the critical section. When there is contention, MCS lock will require one additional write to other processor's locally-accessible shared memory. Both MCS lock and Huang's algorithm uses blocked receive instead of spinning and polling, hence we can reduce memory reference even when there is contention.

## 5 Conclusion

Here we reported our attempt at implementing various fast mutual exclusion algorithms using MPI, and provide some comparisons based on the result of running them.

Our experiment did not yet cover all fast mutual exclusion algorithms. It would be interesting to try to implement other algorithms such as Yang and Anderson's algorithm [13].

There are some points that can be made:

- Our experiment proves that Alur and Taubenfeld's argument, and Michael and Scott's argument that Lamport's algorithm is inefficient in the case of contention is incorrect, since the shared variable *Want*[] can be simply implemented as a word (e.g., integer), if we know the upper bound on the num-

ber of participating processors. Thus it could be useless to have a delay in entry section, which imposes upper bound on memory access speed.

- MCS-based lock algorithms (including Huang's) does not actually have to spin on local memory. They can just wait using blocking receive message passing primitive (although this could have been implemented at the lower level using spins, but I suppose they mostly use interrupts).

## References

- [1] Rajeev Alur and Gadi Taubenfeld. Fast timing-based algorithms. *Distributed Computing*, 10(1):1–10, 1996.
- [2] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 2000.
- [3] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS-V Proceedings: Fifth Symposium on Architectural Support for Programming Languages and Operating Systems, 12–15 October 1992, Boston, MA, USA, ACM SIGPLAN Notices*, volume 27, pages 223–233. ACM Press, September 1992.
- [4] Shiwa S. Fu and Nian-Feng Tzeng. A circular list-based mutual exclusion scheme for large shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):628–639, June 1997.
- [5] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.
- [6] Ting-Lu Huang. Fast mutual exclusion algorithms using read-modify-write and atomic read/write registers. In *International Conference on Parallel and Distributed Systems, 14–16 December, 1998, Taiwan*. IEEE Computer Society Press, December 1998.
- [7] Ting-Lu Huang and Chien-Hua Shann. A comment on “a circular list-based mutual exclusion scheme for large shared-memory multiprocessors”. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):414–415, April 1998.
- [8] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Bell Telephone Laboratories, second edition, 1988.
- [9] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.
- [10] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

- [11] Maged Michael and Michael L. Scott. Fast mutual exclusion, even with contention. Technical Report 460, Department of Computer Science, University of Rochester, June 1993.
- [12] E. Styer. Improving fast mutual exclusion. In *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing, August 10–12, 1992, Vancouver, BC, Canada*, pages 159–168. ACM Press, 1992.
- [13] Jae-Heon Yang and James H. Anderson. Fast, scalable synchronization with minimal hardware support. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing, August 15–18, 1993, Ithaca, NY, USA*, pages 171–182. ACM Press, August 1993.