Efficient Mutual Exclusion Algorithm for High System Congestion

Tsuyoshi SUZUKI † Michiko INOUE † Hideo FUJIWARA †
† Graduate School of Information Science, Nara Institute of Science and Technology

Abstract We propose an efficient mutual exclusion algorithm with respect to remote memory reference(RMR) complexity that measures remote accesses to shared memory. The worst-case RMR complexity for one access to a critical section with N processes has been proven to be $\theta(\log N)$. Though our algorithm has the same worst case RMR complexity, the algorithm becomes efficient with increasing the number of processes executing concurrently. We show the efficiency using queueing theory and simulation. Furthermore, we improve the algorithm so that the elapsed time from some process exits its critical section to the next wanting process enters its critical section is reduced.

1 Introduction

The mutual exclusion problem is a fundamental problem in distributed synchronization problems and solves conflicting access to shared resources. In a mutual exclusion algorithm, each process repeatedly executes four sections idle, entry, critical, and exit in this order. Entry and exit sections have roles to ensure that critical sections are executed exclusively.

Remote memory reference (RMR) complexity is a meaningful measure for algorithms for distributed systems with a shared memory hierarchy. The RMR complexity counts remote memory accesses that involve the interconnect traffic among processes, and represents communication and computation costs of the algorithms.

In the worst-case RMR complexity for N process mutual exclusion algorithms using read and write operations has been investigated. Many algorithms [4, 5, 3] use an $N/2$ leaf binary tree called an *arbitration tree* whose nodes resolve two process mutual exclusion to solve N process mutual exclusion problem. Yang et al. [5] proposed an algorithm with RMR complexity of $O(\log N)$ and space complexity of $O(N \log N)$. Kim et al. [3] optimized the space complexity to $O(N)$ with preserving RMR complexity of $O(\log N)$. Anderson et al. [1] proposed an adaptive mutual exclusion algorithm whose RMR complexity depends on point contention k that is the maximum number of active processes at the same time. Its RMR complexity is $O(min(k, \log N))$, that is, it is efficient when the point contention is low. Attiya et al. [2] proved the lower bound of $\Omega(\log N)$. Therefore Yang's algorithm [5] is optimal with respect to the worst-case RMR complexity.

In this paper, we propose a mutual exclusion algorithm that is efficient in the case where many processes concurrently execute the algorithm. Though our algorithm still has the worst-case RMR complexity of $O(\log N)$, we show the expected RMR complexity is reduced in some high congested situations. We demonstrate the efficiency of the proposed algorithm using queueing theory and simulation.

The rest of this paper is organized as follows. Section 2 defines the model, and Section 3 briefly introduces Yang's algorithm [5]. Section 4 describes the proposed algorithm Tree Skip (TS) and we improve TS to Fast Tree Skip (FTS) in Section 5. Finally we concludes the paper in Section 6.

2 Model

2.1 Shared Memory System

Shared memory system consists of multiple processes and shared memories. The processes execute asynchronously and communicate with each other via the shared memories. The shared memories can be accessed by read and write operations. We consider two types of system with memory hierarchy.

A *distributed shared memory model* (DSM) consists of distributed local shared memories for processes. Each process locally accesses the variables on its local memory and remotely accesses the variables on other processes's local memory.

In a *cache coherent model* (CC), each process has copies of shared variables whose consistency is guaranteed by a coherence protocol. If the cache has the latest value of a shared variable, the access to the variable is local. Otherwise, the remote access is induced.

2.2 Mutual Exclusion Algorithm

Each process that executes a mutual exclusion (ME) algorithm repeatedly executes four sections, *idle*, *entry*, *critical* and *exit* in this order. Each process executes its entry and exit sections to ensure that critical sections are executed exclusively. We assume that the execution time of a CS is finite. Each process does nothing in its idle section (IS), and a process in its IS can start its entry section at any time.

In ME algorithms, entry and exit sections are designed to satisfy the following conditions.

Exclusion: At most one process executes its CS at any time.

Starvation-freedom: Each process that executes its entry section eventually executes its CS.

Termination of exit section: Each process that executes its exit section eventually terminates this section.

We evaluate ME algorithms with RMR complexity. The RMR complexity for ME algorithms is the number of remote memory references for each process during its entry and exit sections before and after each execution of its critical section respectively.

3 Previous Work

3.1 Yang's algorithm

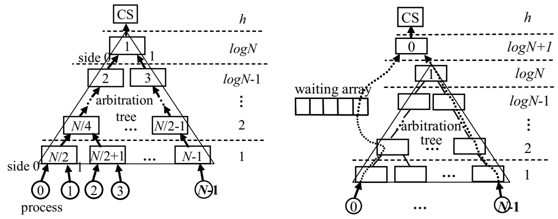
Yang et al. [5] proposed an N process ME algorithm YA with RMR complexity of $O(\log N)$. YA uses a two process ME algorithm (2PME-YA) with constant RMR complexity as a building block.

In YA, the N process ME problem is solved by applying 2PME-YA in a binary tree called an arbitration tree with $N/2$ leaves. Every process is assigned to some leaf of the arbitration tree according to its process ID (Fig. 1(a)), and the process traverses a path from its leaf to the root while executing an entry section of 2PME-YA at each node. Each node has entries from two sides (0 and 1) to distinguish two processes that visit the node. The process can execute its CS when it completes the entry section of 2PME-YA at the root. After execution of the CS, the process then traverses a path from the root to its leaf while executing an exit section of 2PME-YA at the each node. Therefore, the N process ME problem is solved with $O(\log N)$ RMR complexity.

4 Algorithm TS

4.1 Basic Ideas

In ME algorithms using an arbitration tree [4, 5, 3], a process that requires to visit all the nodes in the path from its leaf to the root.



(a) Algorithm YA (b) Algorithm TS

Fig. 1 Algorithm YA and TS

We propose an algorithm using an arbitration tree, where some processes can skip to visit nodes in the path to the root. The proposed algorithm TS (Tree Skip) consists of an arbitration tree with $N/2$ leaves and a waiting array with size of N *. We add a node consists of 2PME-YA at the top to satisfy Exclusion. Let the top node denote the node. Each process checks if it was added to the array at each node in the path. In that case, it waits without traversing the path and then if it was retrieved from the array, it executes two process ME of the top node (Fig. 1(b)). Meanwhile, each process that checked it was not added the array continues to traverse the path. Finally, processes come from the arbitration tree and the waiting array visit this node before entering their CS, and the top node guarantees Exclusion. We call our two process ME algorithm 2PME.

In TS, a process that completed its CS traverses the same path in its entry section in the opposite direction to add other processes that wait at the nodes in the path to the waiting array. Processes are added to and retrieved from the waiting array in FIFO (First In, First Out) order. This guarantees a process added to the array is eventually retrieved from the array, so that the algorithm satisfies Starvation-freedom. We show the code of TS in Fig. 3.

4.2 Algorithm

4.2.1 Shared Variables

Figure 2 describes the shared variables in TS. We use array T, C and P for the same purpose as YA. An array W is the waiting array and *Add* and *Call* indicate the head and tail of the array. *Added*[p] is used to inform p that p is added to the waiting array, where value 1 means that p is added to the array. Value 2 means that p is allowed to leave the waiting array. In DSM, $P[p]$ and *Added*[p] are local variables of the process p.

* We assume N is power of two for simplicity, but the algorithm can be easily adjusted to the general N case.

```
ID : process identifier
const
    L : logN
shared variables
    T[0..N − 1] : integer
    C[0..N − 1][0, 1] : integer initially −1
    P[1..logN + 1][0..N − 1] : (0, 1, 2) initially 0
    W[0..N − 1] : integer initially −1
    Added[0..N − 1] : (0, 1, 2) initially 0
    Add : integer initially N − 1
    Call : integer initially N − 1
```

Fig. 2 Shared Variables in TS

4.2.2 Entry Section

Entry section is from line 3 to 20 in the Fig. 3. In lines 3 to 12, each process executes its entry section of 2PME from its leaf (level 1) to the root (level $\log N$). If the process notices that it is added to the waiting array (line 7) on the way to the root, it skips the remaining nodes and waits until it is allowed to proceed to the top node (line 18). The process completes its entry section after completing its entry section of 2PME-YA at the top node. Let 2PME(n) and 2PME(n, s) denote 2PMEs at a node n and at a node n with a side s, respectively.

4.2.3 Exit Section

Exit section is from line 22 to 39. In this section, each process executes the operations for the waiting array before executing its exit section of 2PME-YA at the top node. Therefore, these operations are executed exclusively, and the waiting array is maintained properly.

In AddtoArray (Fig. 5), each process p adds other processes to the waiting array. Process p checks nodes through the path traversed at its entry section in the reverse order. If there are waiting processes at some nodes, p adds the processes to the waiting array.

In DecisionCall (Fig. 6), a process p determines if there is a process that can proceed to the top node by checking a value of W. Process p decides to allow a process q to proceed to the top node, if other processes that was allowed to proceed to the top node through the waiting array executed their exit section of 2PME-YA at the top node (value -1). DecisionCall just checks the existence of such a node, and it is allowed to proceed later.

Finally, a process p executes its exit sections at the nodes through the path traversed by p in the reverse order.

4.3 Correctness

We prove the correctness of TS. In the proof, $l@p$(Func) means a line l for a process p in a function Func and $l@$(Func) means a line l in a function Func. [$l_1@p$(Func), $l_2@p$(Func)] means lines from

Algorithm 1 TS

```
private variable
    h, node, side, tmpcall, rival : integer
    skip : integer initially −1
    callflg : boolean

1: while true do
2:     idle section;
   /*entry section*/
3:     for h := 1 to L do
4:         node := ⌊(N+ID)/2^h⌋;
5:         side := ⌊(N+ID)/2^(h−1)⌋ mod 2;
6:         Entry2PME(node, side, h)
7:         if Added[ID] > 0 then
8:             await Added[ID] > 1
9:             skip := h;
10:            break;
11:        end if
12:    end for
13:    if skip = −1 then
14:        skip := L;
15:        Entry2PME(0, 1, L + 1)
16:        side := 1;
17:    else
18:        Entry2PME(0, 0, L + 1)
19:        side := 0;
20:    end if
21:    critical section;
   /*exit section*/
22:    AddtoArray(skip)
23:    callflg := DecisionCall(side)
24:    if callflg = true then
25:        tmpcall := Call
26:        rival := W[tmpcall];
27:        W[tmpcall] := −2;
28:    end if
29:    Exit2PME(0, side, L + 1)
30:    if callflg = true ∧ Added[rival] = 1 then
31:        Added[rival] := 2; /*call rival*/
32:    end if
33:    for h := skip down to 1 do
34:        node := ⌊(N+ID)/2^h⌋;
35:        side := ⌊(N+ID)/2^(h−1)⌋ mod 2;
36:        Exit2PME(node, side, h)
37:    end for
38:    skip := −1;
39:    Added[ID] := 0;
40: end while
```

Fig. 3 Algorithm 1 TS

Algorithm 2 Entry2PME(*node,side,h*:integer)

```
private variable
    rival : integer

 1: C[node][side] := ID;
 2: T[node] := ID;
 3: P[h][ID] := 0;
 4: rival := C[node][1 − side];
 5: if rival ≠ −1 then
 6:     if T[node] = ID then
 7:         if P[h][rival] = 0 then
 8:             P[h][rival] := 1;
 9:         end if
10:         await P[h][ID] > 0
11:         if T[node] = ID then
12:             await P[h][ID] > 1
13:         end if
14:     end if
15: end if
```

Fig. 4 Algorithm 2 Entry2PME

Algorithm 3 AddtoArray(*skip*: integer)

```
private variable
    h, node, rival, tmpadd : integer

tmpadd := Add;
for h := skip down to 1 do
    node := ⌊(N+ID)/2^h⌋;
    rival := T[node];
    if rival ≠ ID then
        tmpadd := (tmpadd + 1) mod N;
        if Added[rival] = 0 then
            Added[rival] := 1;
            W[tmpadd] := rival;
        end if
    end if
end for
Add := tmpadd;
```

Fig. 5 Algorithm 3 AddtoArray

Algorithm 4 DecisionCall(*side*: integer)

```
private variable
    tmpcall, precall, rival : integer
    callflg : boolean initially false

 1: precall := Call;
 2: tmpcall := (precall + 1) mod N;
 3: rival := W[tmpcall];
 4: if rival ≥ 0 then
 5:     if side = 0 ∨ W[precall] = −1 then
 6:         Call := tmpcall;
 7:         callflg := true;
 8:     end if
 9: end if
10: if side = 0 then
11:     W[precall] := −1;
12: end if
13: return(callflg);
```

Fig. 6 Algorithm 4 DecisionCall

Algorithm 5 Exit2PME (*node,side,h* :integer)

```
private variable
    rival : integer

1: C[node][side] := −1;
2: rival := T[node];
3: if rival ≠ ID then
4:     P[h][rival] := 2;
5: end if
```

Fig. 7 Algorithm 5 Exit2PME

l_1@p(Func) to l_2@p(Func). In this section, "call" means that to execute 31@(TS) and let n be any natural number.

Termination of exit section obviously holds since there is no waiting loop in the exit sections of TS. Thus, we show the other two properties in TS.

4.3.1 Exclusion

The following condition holds for all the nodes from both sides in YA [5].

Condition 1 ([5])**.** At most one process concurrently executes [1@(Entry2PME(v, s)), 1@(Exit2PME(v, s))] for a pair of node v and side s.

If Condition 1 holds for all the nodes from both sides, Exclusion is satisfied. The arbitration tree in TS includes YA. Thus, Condition 1 holds for nodes except 2PME(0). Furthermore, a process that executes its entry section of 2PME(0, 1) corresponds to a process that executes its CS in YA. Since YA ensures Exclusion, this condition holds for 2PME(0, 1). Thus, Exclusion is satisfied if we just prove that Condition 1 holds for 2PME(0, 0).

First, we prove the following lemma.

Lemma 1. For any process p, other processes are not called in an interval between calling p and 29@p(TS).

Proof. We prove the lemma by induction with respect to the number n of called processes.

For $n = 1$, let p_1 and q_1 be the first called process and a process that calls p_1, respectively. Assume that some processes call in the interval between calling p_1 and 29@p_1(TS). Let r_1 be a process that calls first among these processes. Since r_1 is the second process that calls some process and p_1 is called by the first process q_1 before r_1's call, r_1 executes 2PME(0) with $side = 1$.

Before r_1's call, Condition 1 holds for both sides of 2PME(0), and therefore, [21@(TS),28@(TS)] is executed exclusively. That is, r_1 is the first process that increments $Call$ and turns its $callflg$ into $true$ in DecisionCall after p_1 is called. That is, r_1 executes the interval before p_1 executes the same interval. No process with $side = 0$ initializes W at 11@(DecisionCall) before r_1's call. When r_1 reads $W[precall]$ at 5@(DecisionCall), $W[precall] = -2$ holds and r_1 becomes $false$. This contradicts our assumption. Thus other processes are not called in an interval between calling p_1 and 29@p_1(TS).

Next for $n = k$, let p_h be the hth called process ($h = 1, 2, \cdots, k$). We assume other processes are not called in an interval between calling p_h and 29@p_h(TS). We call this Assumption 1.

For $n = k + 1$, let p_{k+1} be the $(k+1)$th called process. By Assumption 1, Condition 1 holds for 2PME(0, 0) before p_{k+1} is called. Thus, processes exclusively execute [21@(TS), 28@(TS)] until p_{k+1} is called. When p_{k+1} is called, $W[j_{k+1}] = -2$ and $Call = j_{k+1}$ hold, where j_{k+1} is an index of W to which p_{k+1} was added. We assume there are processes that call in an interval between calling p_{k+1} and 29@p_{k+1}(TS). Let r_{k+1} be a process that calls first among the processes. *side* of r_{k+1} is 1 because of Assumption 1. *Call* is only changed by processes that call. Thus, when r_{k+1} executes 5@r_{k+1}(DecisionCall), r_{k+1} reads $W[j_{k+1}]$ since $precall = j_{k+1}$ holds. 11@(DecisionCall) is executed to change a value of $W[j_{k+1}]$ into -1. But, there is no process that executes the line in an interval between calling p_{k+1} and 5@r_{k+1}(DecisionCall). So the value of $W[j_{k+1}]$ is not changed into -1. Thus, this contradicts our assumption and any other processes are not called in an interval between calling p_{k+1} and 29@p_{k+1}(TS). □

Lemma 2. For 2PME(0, 0) at most one process concurrently executes [18@(TS), 29@(TS)].

Processes that execute 2PME(0, 0) are called at 31@(TS), and then execute [18@(TS), 29@(TS)]. Lemma 1 implies no other process starts [18@(TS), 29@(TS)] while other process is executing this interval with side of 0. This means the lemma holds.

Thus, 2PME(0, 0) satisfies Condition 1.

Theorem 1. TS ensures Exclusion.

4.3.2 Starvation-freedom

We prove Starvation-freedom of TS. All waiting loops in entry sections of TS are eventually completed iff TS ensures this property. We already prove Exclusion property, so we use the following property.

Property 1. At most one process executes [21@(TS) , 28@(TS)] concurrently.

For any process p, waiting loops in its entry section are as follows.

Wait 1. 10@p(Entry2PME) (**await** $P[h][p] > 0$)

Wait 2. 8@p(TS) (**await** $Added[p] > 1$)

Wait 3. 12@p(Entry2PME) (**await** $P[h][p] > 1$)

Entry2PME is the same as 2PME-YA. We add skip mechanism outside the code of 2PME-YA. The skip mechanism just stops process's proceeding to the next 2PME in the arbitration tree, and the skipped process will return the Exit2PME, where the process stopped the proceeding.

Since Wait 1 and 3 are in Entry2PME and only Wait 2 is a waiting loop outside Entry2PME, if Wait 2 is eventually completed, TS ensures Starvation-freedom.

We prove Wait 2 is eventually completed. First, we show the updating rule of $Added[p]$. There are three operations that update $Added[p]$.

Update 0 at 39@p(TS) (p initializes it)

Update 1 at 8@(AddtoArray) (p is added to the array)

Update 2 at 31@(TS) (p is called)

We prove the following lemma to prove Wait 2 is eventually completed.

Lemma 3. If Update 2 is executed after any process p writes p to T at 2@(Entry2PME), $Added[p]$ is not updated until 8@p(TS) is completed.

Proof. Since [30@(TS), 31@(TS)] implies that Update 2 is executed if $Added[p] = 1$, Update 1 is executed before Update 2. Let q be a process that executes the Update 1. The process q executes Update 1 in its AddtoArray, if q finds $T[n] = p$ for some node n. In TS, q first executes Entry2PME at some nodes, and then visits these nodes again in AddtoArray, and finally executes Exit2PME at these nodes.

Therefore, q finds $T[n] = p$ after q completes Entry2PME(n) and before starts Exit2PME(n). In Entry2PME(n), q writes q to $T[n]$, and therefore p writes p to $T[n]$ in Entry2PME(n) after q writes q to $T[n]$. Since Condition 1 holds for 2PME(n) after p writes p to $T[n]$, no process executes 2PME(n) with the same side with q before q executes Exit2PME(n), and no process executes 2PME(n) with the same side with p before p completes Entry2PME(n). Therefore, no process updates $T[n]$ before q starts Exit2PME(n) or p completes Entry2PME(n).

We next show that p does not complete Entry2PME(n) before q starts Exit2PME(n). For p to complete Entry2PME(n) before q starts Exit2PME(n), p completes the waiting loops at 10@(Entry2PME(n)) and 12@(Entry2PME(n)) since p always finds $T[n] = p$. To complete 12@(Entry2PME(n)), $P[h][p] \geq 2$ holds. However, $P[h][p]$ is updated to 2 only at 4@(Exit2PME(n)) executed by q. Therefore, p does not complete Entry2PME(n) before q starts Exit2PME(n), and therefore, q executes Update 1 in AddtoArray before p completes Entry2PME(n). This implies that p finds $Added[p] > 0$ at 7@(TS) after p completes Entry2PME(n). Then p waits until $Added[p] = 2$

holds at 8@(TS). Since Update 0 is executed only by p itself at 39@(TS), Update 0 is not executed after Update 2 before p completes 8@(TS). Moreover, Update 1 is executed only when $Added[p] = 0$ at 8@(AddtoArray), since AddtoArray is executed exclusively by Property 1. Therefore, after Update 2, $Added[p]$ is not updated until 8@(TS) is completed. □

Corollary 1. After Update 1, p does not execute Update 0 until 8@p(TS) is completed.

This corollary is used in the next subsection.

Furthermore, we prove the following lemma for Wait 2.

Lemma 4. Any process p that is added to the waiting array W eventually completes Wait 2.

Proof. Lemma 3 means that any called process eventually complete Wait 2. Therefore, Lemma 4 holds if process p that is added to the waiting array is eventually called. We prove this by induction with the respect to the number n of called processes.

For $n = 1$, let p_1 be a process that is added to W first. There is no process that execute their entry section of 2PME(0, 0) before p_1 executes its entry section of 2PME(0, 0). Thus, a process q that adds p_1 to W visits 2PME(0, 1). q writes p_1 to $W[0]$ at 9@q(AddtoArray). There is no process that changes a value of W or $Call$ before q executes 28@q(TS) by Property 1. Thus, the condition is true at 5@q(DecisionCall) and $callflg = true$ holds since $W[N-1] = -1$ (initial value) holds. Process q reads p_1 from $W[0]$ and sets $rival = p_1$ at 26@q(TS) and writes 2 to $Added[p_1]$ at 31@q(TS). Therefore, p_1 eventually completes Wait 2 (by Lemma 3).

Next, for $n = k$ let p_k be the kth process that is added to W and we assume p_k eventually completes Wait 2.

For $n = k+1$, let p_{k+1} be the $(k+1)$th process that is added to W. Let j be an index of W to which p_{k+1} is added and $j_{pre} = (j-1) \bmod N$. AddtoArray is exclusively executed by Property 1, so p_k is written to $W[j_{pre}]$.

When p_{k+1} is written to $W[j]$, a value of $W[j_{pre}]$ is -1, -2 or p_{k+1}.

(i) Case $W[j_{pre}] = -1$ (it is initialized.)

In the case, p_k writes -1 to $W[j_{pre}]$ at 11@p_k(DecisionCall) before p_{k+1} is added to W. Thus, another process r adds p_{k+1} to W. After r executes this operation, no other process updates W or $Call$ before r executes 28@r(TsS) by Property 1. Thus, r reads $W[j_{pre}] = -1$ at 5@(DecisionCall), and therefore, the condition is true at 5@r(DecisionCall) and $callflg = true$ holds. Process r reads p_{k+1} from $W[j]$ at 26@r(TS). Then, r writes 2 to $Added[p_{k+1}]$ at 31@r(TS). p_{k+1} eventually completes Wait 2 by Lemma 3.

(ii) Case $W[j_{pre}] = -2$ or p_{k+1}

In this case, p_k has not executed 11@(DecisionCall) when p_{k+1} is added to W, since $W[j_{pre}]$ is initialized to -1 by p_k at 11@(DecisionCall). The interval [21@(TS) , 28@(TS)] is executed exclusively by Property 1, and the interval includes AddtoArray and DecisionCall. Therefore, p_k is added before p_k reads $W[j]$ at 3@(DecisionCall). While $W[j_{pre}] \neq -1$, no process calls p_{k+1} and updates $W[j]$, Therefore, p_k reads p_{k+1} from $W[j]$. The condition 5@p_k(DecisionCall) is true since p_k's $side = 0$. Therefore, p_k eventually calls p_{k+1}. p_{k+1} eventually completes Wait 2 by Lemma 3. □

Next, we prove the following lemma using Lemma 4.

Lemma 5. Wait 2 is eventually completed.

Proof. If process p executes 8@p(TS), p was added to the waiting array before then. Therefore, by Lemma 4, p eventually completes Wait 2. □

TS satisfies Starvation-freedom.

Theorem 2. TS ensures Starvation-freedom.

4.4 Evaluation of TS

We evaluate RMR complexity for the proposed algorithm. We show the worst case complexity, and then give two kinds of analysis using queueing theory and simulation.

4.4.1 Evaluation by the number of nodes

We first show the relation between RMR complexity and the number of nodes that a process visits in its entry section.

Lemma 6. RMR complexity of TS for N processes is proportional to the number of nodes that a process visits in its entry section.

Proof. In TS, a process executes its entry section of 2PME, one iteration of AddtoArray and its exit section of 2PME at each node. These are able to be executed with constant remote memory accesses. Process p executes other operations while it is in the waiting array in [7@(TS), 8@(TS)]. In the case of DSM, p executes no remote memory access since $Added[p]$ is local to p. In the case of CC, Corollary

1 means that once *Added*[p] is set to 1, it is stable to 1 until it is set to 2. This implies that p executes constant remote memory accesses in the waiting array. Therefore, other procedures (waiting the array (8@(TS)), allowing another process to skip nodes ([30@(TS), 32@(TS)]) and DecisionCall) are executed with constant remote memory accesses too. Thus, TS's RMR complexity is proportional to able to the number of nodes that a process visits in its entry section. □

4.4.2 Worst Case Complexity

Each process traverses the path from its leaf to the root. Therefore the process visits at most $\log N+1$ nodes.

Theorem 3. RMR complexity of TS for N processes is $O(\log N)$

4.4.3 Analysis by Queueing Theory

In TS, the more processes concurrently execute their entry and exit sections, the more frequently they skip the nodes of the tree. We evaluate this using queueing theory.

We evaluate the average case RMR complexity in the case where all the processes behave uniformly. We use $M/M/1(1)$ queueing system that has negative exponential interarrival times and service times with a single server and no waiting queue. Let λ and μ are an average arrival rate and average service rate, respectively. In $M/M/1(1)$ system, probability P_{ocpy} that the system is in service and probability P_{empty} that the system is not in service are given as follows.

$$P_{ocpy} = \frac{\lambda}{\lambda+\mu} \tag{1}$$

$$P_{empty} = \frac{\mu}{\lambda+\mu} \tag{2}$$

Service Model We consider a service for each side at each node, where a service at level l starts at 2@(Entry2PME) and ends at 4@(AddtoArray) with $h = l$ (Fig. 8). We consider the case where the average interarrival rate and the average service rate are the same for the same level. That is, a service for a process p at level l includes services for p at level $l'(l' > l)$. Let λ_k and μ_k denote the average interarrival rate and the average service rate for a service at level k, respectively.

We analyze the case where the interarrival and service times for at level 1 (leaf level) are negative exponentially distributed. We apply $M/M/1(1)$ system to the services.

First, we calculate λ_k ($k = 2, \cdots \log N$) (Fig. 9). We consider a process p starts the service at level $k-1$ at some node. If no process is in service at the same node, p is not added to the waiting array and will start the service at level k. Since each node has entries from two sides, λ_k is derived as follows.

$$\lambda_k = 2\lambda_{k-1}P_{empty,k-1} \tag{3}$$

Next, we calculate μ_k. Assume that a process starts a service at level k. In this case, when the process executes 2@(Entry2PME) at level $k-1$, if no process is in service at the same node, the process completes Entry2PME at level $k-1$ without waiting other processes's operations at 10@(Entry2PME) or 12@(Entry2PME), since waiting process u is eventually added to the waiting array and never proceeds to a node at level k. Therefore, the difference of service time between levels $k-1$ and k is one iteration of the loop at [3@(TS), 3@(TS)] and one iteration of the loop at [2@(AddtoArray), 12@(AddtoArray)]. Therefore, the time difference is considered to be independent of levels. Let m denote this time difference. Thus, μ_k is derived as follows.

$$\frac{1}{\mu_k} = \frac{1}{\mu_{k-1}} - m \tag{4}$$

We can obtain $P_{ocpy,k}$ and $P_{empty,k}$ from the equations (3) and (4) and obtain the expected number E of nodes that a process visit in its entry and exit sections as follows.

$$E = \sum_{k=1}^{\log N-1} \{P_{stop,k}\cdot(k+1)\} + P_{pass,\log N}\cdot(\log N+1) \tag{5}$$

where

$$P_{pass,k} = P_{empty,1}\cdot P_{empty,2}\cdot \ldots \cdot P_{empty,k} \tag{6}$$

and

$$P_{stop,k} = P_{ocpy,k}\cdot P_{pass,k-1} \tag{7}$$

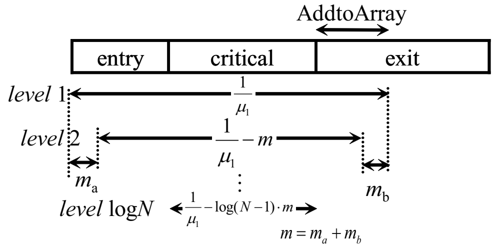


Fig. 8 The service at each level

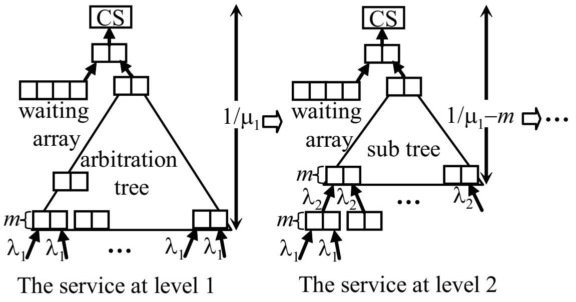


Fig. 9 The recursive calculus of λ_k and $\frac{1}{\mu_k}$

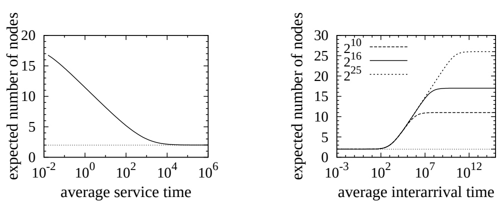
Table 1 Case Study Prameters

Parameter	Value
Number of Processes	65536
Average Interarrival Time	1000
Average Service Time	1000
m	0.001

Case Study We use values of parameter shown in Table 1 as default, and consider cases by varying parameters.

Figure 10(a) shows the case where the average service time is varied. The number of visited nodes is reducing with increasing the service time. We consider this is because the longer the service time is, the more frequently processes skip.

Figure 10(b) shows the case where the average interarrival time and the number of processes N are varied. When the interarrival time is long, the numbers of visited nodes close to $\log N + 1$ and the numbers are reducing and converge to 2 with reducing the interarrival time. We consider this is because the shorter this time is, the more congested the system is and the more frequently processes skip. Furthermore, this result shows when the system is much congested, the expected number of visited nodes does not depend on the number of processes.



(a) Service Time (b) Interarrival Time

Fig. 10 Expected Number of Visited Nodes

4.4.4 Analysis by Simulation

Simulation Setup We evaluate the performance of TS by simulation. In the simulation, execution times of IS, CS, one remote memory access, one local memory access and one local operation are different for every processes and their average times among processes are varied in the range of ±100% of the values in Table (2). These execution times for each process is varied in the range of ±50% of the averages. We set a bandwidth that is the maximum number of processes accessing shared memories concurrently as in Table (2).

We measure RMR complexity and the execution time that is taken in entry and exit sections for one CS. We simulate TS until each process executes its CS 1,000 times. Since the performance only after the system is in equilibrium is meaningful, we get the data after every process enters its CS 10 times.

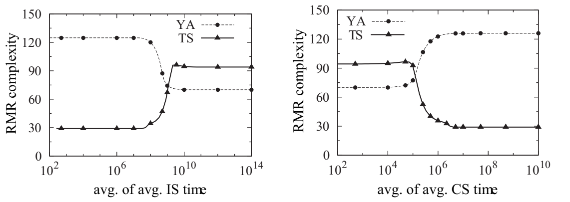
Table 2 Simulation Setup Prameters

parameter		value
number of processes		16384
avg. of avg. time	IS	10^{10}
	CS	5000
	remote memory access	500
	local memory access	5
	local operation	1
bandwidth		16384

Results We show the results for only DSM, since the results for CC are similar to the case of DSM. Figure 11 (a) shows the RMR complexity when the avg. of avg. IS time is varied. It is observed that the shorter IS time is, the fewer RMR complexity is. Also, Fig 11 (b) shows the result when the avg. of avg. CS time is varied. It is observed that the longer CS time is, the fewer RMR complexity is. We consider the more congested system (the sorter IS time or the longer CS time) is, the more processes try to execute their CS and less RMR complexity is. However, when the system is not congested (long IS execution time or short CS execution time), TS has more RMR complexity than YA. This is because TS always has overhead to maintain the waiting array even if there are no process to be added to the array.

We further examine the performance at high system congestion. We change the avg. of avg. times of IS and CS into 0 and 50,000, respectively. Figure 12 shows the RMR complexity when the number of processes is varied. We find that TS's RMR complexity does not depend on the number of processes while YA's RMR complexity depends on it.

Finally, we evaluate an actual execution time for the entry and exit sections. Figure 13 shows the execution time of entry and exit sections before and after each execution of CS. In the case where the bandwidth is narrow, the execution time of TS is shorter than YA, but it is longer than YA when the bandwidth is wide. We consider this is because each process executes AddtoArray and DecisionCall exclusively in TS, and no process can start its CS until the process completes these procedures. We consider such waiting time has a significant influence for the execution time.



(a) IS Time (b) CS Time

Fig. 11 RMR complexities and IS/CS execution time

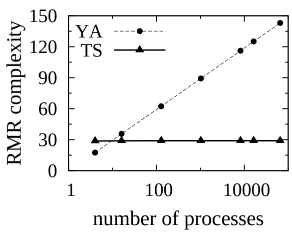


Fig. 12 RMR complexity and the number of processes

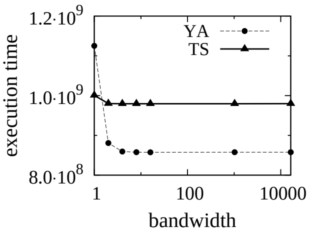


Fig. 13 Execution time and the bandwidth

5 Improving TS to FTS

In TS, each process has at least 20 remote memory accesses at the beginning of its exit section before the next process starts its CS. That is a main reason why TS has longer execution time. We modify TS to FTS(Fast TS) to resolve this problem.

5.1 Basic Ideas

FTS has two key ideas. First, we separate the privilege to maintain the waiting array from the privilege to execute CS. A process that completes its CS first releases the privilege for CS, and then starts to aquire the privilege to maintain the waiting array. Second, we divide the waiting array into two for skipping process to execute CS before the waiting array is maintained.

Figure 14 shows the overview of FTS. To execute CS, processes from two waiting arrays and an arbitration tree compete in three process ME 3PME(0). A process leaves 3PME(0) at the beginning of its exit section so that the next process can start CS soon. Then, the process starts to aquire the privilege to maintain the waiting arrays. At that time, the next process from the same waiting array might catch up on the process, therefore at most 5 processes only join the competition. This modification enable each process to have only at least 4 remote memory accesses at the beginning of its exit section before the next process starts its CS.

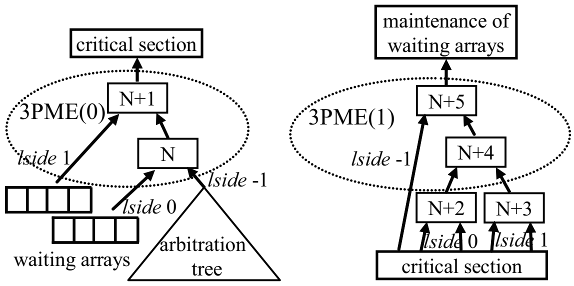


Fig. 14 Algorithm FTS

5.2 Algorithm

5.2.1 Shared Variables

The shared variables in FTS are shown in Fig. 15. These variables are almost the same as TS. The differences between FTS and TS are lengths of T, C and P. Theses arrays are longer to add five nodes ($N+1, N+2, \cdots, N+5$). Furthermore the waiting array W is changed into the two dimensional array and *Call* and *Add* also double to divide the waiting array into two.

5.2.2 Entry Section

An additional modification of each entry section is putting an additional node at the highest level. In TS, two processes, one from the waiting array and the other from the root, finally contend to enter their CS. Meanwhile, three processes, two from the two waiting arrays and one from the root, contend in FTS. Thus, each process executes three process mutual exclusion (3PME(0)) on the root with *lside* (line 20@(FTS)). 3PME(0) is consist of

ID : *process identifier*
const
L : $logN$
shared variables
$T[1..N+5]$: **integer**
$C[1..N+5][0,1]$: **integer** *initially* -1
$P[1..logN+5][0..N-1]$: $(\mathbf{0},\mathbf{1},\mathbf{2})$ *initially* 0
$W[0..\frac{N}{2}-1][0,1]$: **integer** *initially* -1
$Added[0..N-1]$: **integer** *initially* -2
$Add[0,1]$: **integer** *initially* $\frac{N}{2}-1$
$Call[0,1]$: **integer** *initially* $\frac{N}{2}-1$
$Preside[0,1]$: **integer** *initially* 0

Fig. 15 Shared Variables in FTS

two 2PMEs (Fig. 14). A process from the root executes 3PME(0) with $lside = -1$. Processes that executes [9@(FTS), 10@(FTS)] get value $lside$ (0 or 1) to identify two processes that skip nodes from each waiting array.

5.2.3 Exit Section

First, each process releases 3PME(0) in its exit section to allow other processes to execute CS immediately (line 22@(FTS)). 3PME(0) is constructed as shown in Fig. 14. Thus, in the case where a process q that skips nodes from an array is waiting at 2PME($N+1$), q can immediately enter CS after another process p that terminates its CS releases the node. In this case, only at least 4 remote memory accesses are taken after p executes its exit section before q enters its CS.

Next, the process executes another three process mutual exclusion (3PME(1)) to acquire the privilege to maintain the arrays (line 31@(FTS)). The procedures to maintain the arrays are [32@(FTS), 39@(FTS)]. A side of 3PME(1) is corresponding to a side of 3PME(0) ($lside$). However, once a process calls the next process from the waiting arrays, the called processes might catch up on the process while contending 3PME(1). To avoid that two processes concurrently execute 3PME(1) with the same $lside$, we add two nodes ($N+2$ and $N+3$). Processes decide a side of these additional nodes in [26@(FTS), 28@(FTS)]. The procedures invert the value of *side* if the procedure are executed exclusively. The process that acquired this privilege executes AddtoArray2 and CallProcess.

AddtoArray2 (Fig. 17) is a modification of AddtoArray to add processes to the two waiting arrays alternately. Processes from a waiting array W_1 set $rside$ to side of another array W_2, and add some process to W_2 first. Then the processes are added to the two waiting arrays alternately to invert a value of *addside* at line 11@(AddtoArray2).

CallProcess (Fig. 18) is executed to check if some process skips nodes from an array and to allow the process to skip nodes. CallProcess is executed two times since there are the two waiting arrays. Finally, the process executes its exit section of 3PME(1) (and 2PME(N+2 or N+3), if necessary) to release this privilege. Then, the process executes its exit section of 2PME at the nodes in the arbitration tree ([44@(FTS), 48@(FTS)]).

5.3 Correctness

We prove the correctness of FTS. In this section, "call" means that to execute 8@(CallProcess) and let n is any natural number.

5.3.1 Exclusion

We show Exclusion property of FTS. Let p_{k+1} is the $(k+1)$th called process. First, we prove the following lemmas.

Condition 2. At most one process concurrently executes [32@(FTS), 39@(FTS)] before p_{k+1} is called if the following conditions hold.

ME1. At most one process concurrently executes [26@(FTS), 28@(FTS)] from each $lside$ $i (i = 0, 1)$ before p_{k+1} is called.

ME2. At most one process concurrently executes [29@(FTS), 42@(FTS)] for each pair of $lside$ $i (i = 0, 1)$ and $side$ $j (j = 0, 1)$ before p_{k+1} is called.

ME3. Any other processes are not called with $callside$ of i between calling p_h with $callside$ of i and 28@p_h(FTS), where let p_h be the hth called process $(h = 1, 2 \cdots, k)$.

Proof. The lemma holds if Condition 1 holds for the node $N+5$ (Fig. 14). First, we show the condition for the left side ($lside = -1$) at the node. Condition 1 holds for every node in the arbitration tree and holds for the left side at the nodes N and $N+1$ by ME3. Thus, Condition 1 holds for the left side at the node $N+5$. Next, we show Condition 1 for the right side at the node $N+5$. The conditions ME1 and ME2 imply that Condition 1 holds for the nodes $N+2$ and $N+3$. Thus, any other process that executes [32@(FTS), 39@(FTS)] before p_{k+1} is called exclusively execute this interval. □

Lemma 7. ME1 holds if ME3 holds.

This obviously holds.

Lemma 8. ME2 holds if ME1 and ME3 holds.

Proof. We assume the contrary that two or more processes concurrently execute [29@(FTS), 42@(FTS)] with $lside$ i and $side$ j. Let s_1 and s_2 be processes that are called first and second among the processes, respectively. By ME1, since [26@(FTS), 28@(FTS)] inverts the value of $side$,

Algorithm 6 FTS

```
private variable
    h, node, side, lside, precall, rside : integer
    skip : integer initially -1
    callflg : boolean

while true do
    idle section;
    /*entry section*/
    for h := 1 to L do
        node := ⌊(N+ID)/2^h⌋;
        side := ⌊(N+ID)/2^(h-1)⌋ mod 2;
        Entry2PME(node, side, h)
        lside := Added[ID];
        if lside > -2 then
            while lside = -1 do
                lside := Added[ID];
            end while
            skip := h;
            break;
        end if
    end for
    if skip = -1 then
        skip := L;
        lside := -1;
    end if
    Entry3PME(0, lside)
    critical section;
    /*exit section*/
    Exit3PME(0, lside)
    if lside = -1 then
        rside := 0;
    else
        rside := 1 - lside;
        side := 1 - Preside[lside];
        Preside[lside] := side;
        Entry2PME(N + 2 + lside, side, L + 3)
    end if
    Entry3PME(1, lside)
    AddtoArray2(skip, rside)
    precall := Call[rside];
    CallProcess(precall, lside, rside)
    precall := Call[1 - rside];
    CallProcess(precall, lside, 1 - rside)
    if lside ≥ 0 then
        W[precall][lside] := -1;
    end if
    Exit3PME(1, lside)
    if lside ≥ 0 then
        Exit2PME(N + 2 + lside, side, L + 3)
    end if
    for h := skip down to 1 do
        node := ⌊(N+ID)/2^h⌋;
        side := ⌊(N+ID)/2^(h-1)⌋ mod 2;
        Exit2PME(node, side, h)
    end for
    skip := -1;
    Added[ID] := -2;
end while
```

Fig. 16 Algorithm 6 FTS

Algorithm 7 AddtoArray2
($skip, addside$: **integer**)

```
private variable
    h, node, rival, tmpadd[0,1] : integer

tmpadd[0] := Add[0];
tmpadd[1] := Add[1];
for h := skip down to 1 do
    node := ⌊(N+ID)/2^h⌋;
    rival := T[node];
    if rival ≠ ID then
        if Added[rival] = -2 then
            tmpadd[addside] := (tmpadd[addside] + 1) mod N/2;
            Added[rival] := -1;
            W[tmpadd[addside]][addside] := rival;
            addside = 1 - addside;
        end if
    end if
end for
Add[0] := tmpadd[0];
Add[1] := tmpadd[1];
```

Fig. 17 Algorithm 7 AddtoArray2

Algorithm 8 CallProcess
($precall$, $lside$, $callside$ **:integer**)

```
private variable
    tmpcall, callside : integer
    rival : integer initially -1

if W[precall][callside] = -1 ∨ (lside = callside) then
    tmpcall := (precall + 1) mod N/2;
    rival := W[tmpcall][callside];
    if rival ≥ 0 then
        if Added[rival] = -1 then
            Call[callside] := tmpcall;
            W[tmpcall][callside] := -2;
            Added[rival] := callside; /*call rival*/
        end if
    end if
end if
```

Fig. 18 Algorithm 8 CallProcess

in order to execute 29@(FTS) with the same *lside* and *side*, these should be executed by another process t between s_1 and s_2. Since process s_2 violates ME2, Condition 2 holds before s_2 is called. Thus, processes exclusively execute [32@(FTS), 39@(FTS)] before s_2 is called. $Call[i] = j_{s1}$ and $W[j_{s1}][i] = -2$ hold in the interval between calling s_1 and 38@s_1(FTS), where j_{s1} is an index at which s_1 is added to W. Thus, t is called by s_1 at 36@s_1(FTS) or another process after s_1 executes 38@s_1(FTS). t executes its entry section of 2PME-YA at the node $N+2$ or $N+3$. t does not terminate this section before s_1 releases the node at 42@s_1(FTS). Then, s_2 is called also. In the same way of calling t, s_2 is called by t at 36@t(FTS) or another process after t executes 38@t(FTS). But t does not complete to execute 29@t(FTS) before s_1 executes 38@s_1(FTS). This contradicts our assumption. □

Lemma 9. There is no process that is called with *callside* i between calling p and 28@p(FTS).

Proof. We prove the lemma by induction with respect to the number n of called processes.

For $n = 1$, let p_1, i_p and q_1 be the first called process, *callside* of p_1 and a process that calls p_1, respectively. Processes only with $lside = -1$ execute 20@(FTS) (Entry3PME(0)) and 31@(FTS) (Entry3PME(1)) before q_1 calls p_1. That is, the processes do not execute Entry2PME(0) from $lside = 0$ or 1. Condition 1 holds for the nodes in the tree. Thus, processes that execute [32@(FTS), 39@(FTS)] before q_1 calls p_1 exclusively execute this interval.

Thus, when q_1 calls p_1, $Call[i_p] = 0$ and $W[0][i_p] = -2$ hold. We assume the contrary there are processes that calls with *callside* of i_p between calling p_1 and 28@p_1(FTS). Let r_1 be a process that calls first with *lside* of -1 among the processes. $Call[i_p]$ is updated only at 6@(CallProcess) and each process updates it if the process calls some process. Thus, when r_1 executes CallProcess, $Call[i_p] = 0$ holds and r_1 reads $W[0][i_p]$ at 1@r_1(CallProcess). Each process executes 38@(FTS) with *lside* of i_p to write -1 to $W[0][i_p]$. But, no process executes 38@(FTS) with *lside* of i_p before r_1 executes 1@r_1(CallProcess). Thus, A value of $W[0][i_p]$ is not changed into -1 before then, and the condition is false at 1@r_1(CallProcess). Thus, this contradicts our assumption.

For $n = k$, we assume ME3.

For $n = k+1$, let p_{k+1} be the $(k+1)$th called process. By Condition 2 and Lemma 7 and 8, when p_{k+1} is called, $W[j_{k+1}][i] = -2$ and $Call[i] = j_{k+1}$ hold. We assume the contrary there are processes which calls with *callside* of i between calling p_{k+1} and 28@p_{k+1}(FTS). Let r_{k+1} be a process that calls first among the processes.

When r_{k+1} executes CallProcess, $Call[i] = j_{k+1}$ holds and r_{k+1} reads $W[j_{k+1}][i]$ at 1@r_{k+1}(CallProcess). No process executes 38@(FTS) with *lside* of i before r_{k+1} executes 1@r_{k+1}(CallProcess). Thus, A value of $W[0][j_{k+1}]$ is not changed into -1 before then, and the condition is false at 1@r_{k+1}(CallProcess). This contradicts our assumption.

Thus, between calling any process p with *callside* of i and 28@p(FTS), any other processes are not called with *callside* of i. □

Corollary 2. At most one process concurrently executes [32@(FTS), 39@(FTS)].

Condition 1 holds for all the nodes in the tree. Thus, Exclusion is satisfied if Condition 1 holds for 3PME(0). The topology of 3PME(0) is shown in Fig. 14. Processes that execute their entry sections of 2PME-YA at 2PME($N+5$) from *lside* of -1 correspond to processes that pass the node of the root in the arbitration tree. Thus, the condition holds for 3PME(0, -1).

Lemma 10. For 3PME(0) at most one process concurrently executes from each *lside* i ($i = 0, 1$) [20@(FTS), 22@(FTS)].

Processes that execute 3PME(0, i) are called at 8@(CallProcess), and then execute [20@(FTS), 22@(FTS)]. Lemma 9 implies no other process starts [20@(FTS), 22@(FTS)] while other process is executing this interval with $lside = i$. This means the lemma holds.

Thus, Condition 1 holds for all the nodes and FTS satisfies Exclusion.

Theorem 4. FTS ensures Exclusion.

5.3.2 Starvation-freedom

We prove Starvation-freedom of FTS. For that purpose, we show all waiting operations in each entry section of FTS eventually are completed.

For any process p, waiting operations in its entry section are as follows.

Wait 1. 10@p(Entry2PME) (**await** $P[h][p] > 0$)

Wait 2. [9@p(FTS), 10@p(FTS)] (**await** $Added[p] \neq -1$)

Wait 3. 12@p(Entry2PME) (**await** $P[h][p] > 1$)

If Wait 2 is eventually completed, FTS satisfies Starvation-freedom by the same reason as the reason of TS.

Lemma 11. Wait 2 is eventually completed.

Wait 2 is executed when processes are added to the waiting arrays and is completed when the processes are called by some process. FTS has the two waiting arrays. A process checks if the process calls some process from each of the arrays at 34@(FTS) and 36@(FTS). Since the mechanism to add and call processes for each of the arrays in FTS is the same as TS's, the lemma holds by almost the same proof in Lemma 5.

Theorem 5. FTS ensures Starvation-freedom.

5.3.3 Termination of exit section

There are waiting operations in each exit section of FTS. The operations is executed at the nodes of $N+2$, $N+3$, $N+4$ and $N+5$ in Entry2PME. Each process visits the nodes to acquire the privilege to maintain the waiting arrays. Since there is no waiting loop in the procedures to maintain the arrays, the operations are eventually completed obviously.

Theorem 6. FTS ensures Termination of exit section.

5.4 Evaluation of FTS

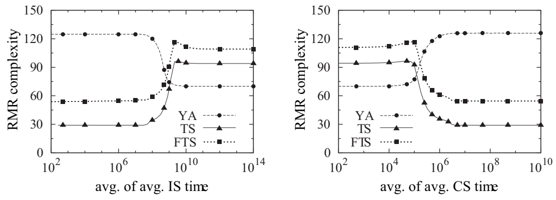
We evaluate the RMR complexity and the execution time of FTS by simulation. We basically use the parameter in Table 2 and vary the avg. of avg. times of IS and CS into 0 and $50,000$, respectively in Fig.20 and 21.

Figure 19 and 20 show the RMR complexities. These have the same tendency as TS. FTS has more RMR complexities than TS in all cases. This is because FTS has more overhead than TS to maintain the two waiting arrays. But we consider that its overhead is constant for number of processes as shown in Fig. 20.

Figure 21 shows the execution times for the cases when (a) average CS time are different among processes, and (b) average CS time are common to processes. These results show the execution time is reduced than TS. Furthermore, we find that the execution time is improved to almost the same value as YA in Fig. 21 (b). We consider this is because each process has enough CS time corresponding to maintain the arrays and therefore the elapsed time from a process terminates its CS to another process enters its CS is almost the same time as YA.

6 Conclusion

We proposed the mutual exclusion algorithm for distributed system with shared memory hierarchy. The algorithm TS allows processes to skip



(a) IS Time (b) CS Time

Fig. 19 RMR complexity and IS/CS execution time

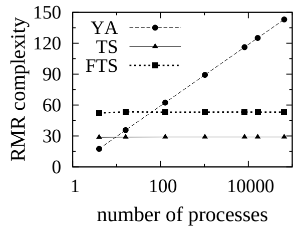
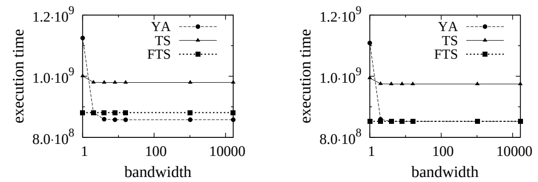


Fig. 20 RMR complexity and the number of processes

the node in the arbitration tree. Its worst-case RMR complexity of $O(\log N)$ is optimal. The proposed algorithm is efficient with respect to RMR complexity when many processes execute the algorithm concurrently. We demonstrated the efficiency by queueing theory and simulation. These results show its RMR complexity is close to $O(1)$ in the case of high system congestion.

Furthermore we improved the algorithm to reduce the actual execution time, In the improved algorithm FTS, for the purpose of reducing the number of remote memory accesses between some process completes its CS and some process executes its CS, we separate the privilege to maintain the waiting array from the privilege for CS and divide the waiting array into two. We demonstrated the efficiency by simulation and the result showed its execution time is almost the same time as YA when each process has enough CS time with maintaining the characteristic of TS.

Though the proposed algorithms have the space



(a) Different avg. CS time (b) Constant avg. CS time

Fig. 21 Execution time and the bandwidth complexity of $O(N \log N)$, the idea [3] to reduce the complexity is applicable, and it can be improved to $O(N)$.

The future work is to propose algorithms that are efficient in the both cases of low and high congestions.

References

[1] J.H. Anderson and Y.J. Kim. Adaptive Mutual Exclusion with Local Spinning. *Distributed Computing: 14th International Conference, DISC 2000, Toledo, Spain, October 2000: Proceedings*, 2000.

[2] H. Attiya, D. Hendler, and P. Woelfel. Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. *Proceedings of the fourtieth annual ACM symposium on Theory of computing*, pages 217–226, 2008.

[3] Y.J. Kim and J.H. Anderson. A space-and time-efficient local-spin spin lock. *Information Processing Letters*, 84(1):47–55, 2002.

[4] G.L. Peterson and M.J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 91–97. ACM New York, NY, USA, 1977.

[5] J.H. Yang and J.H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.