共有メモリシステムにおける調停木スキップ相互排除アルゴリズム

鈴木 健司 †， 井上 美智子 †， 藤原 秀雄 †
† 奈良先端科学技術大学院大学 情報科学研究科

概要 共有メモリへの遠隔操作の回数を評価尺度とするRMR(Remote Memory Reference) 計算量に関して効率の良い相互排除アルゴリズムを提案する．相互排除の最悪RMR計算量は1回の共有資源の獲得のためにプロセスが行うRMRの回数である．N プロセス相互排除アルゴリズムの最悪RMR計算量は $\theta(\log N)$ であることが知られている．我々は最悪時計算量が $\theta(\log N)$ であるが，多くのプロセスが同時に資源獲得を行う場合に効率の良いアルゴリズムを提案し，その効率性を待ち行列理論とシミュレーションの二つの手法を用いて示す．さらにアルゴリズムを計算処理時間に関して改善を行う．

Mutual Exclusion Algorithm with Skipping Arbitration Tree

Tsuyoshi SUSUKI † Michiko INOUE † Hideo FUJIWARA †
† the 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, 6, 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. [6] 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 [6] 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 [6]. 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' 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.

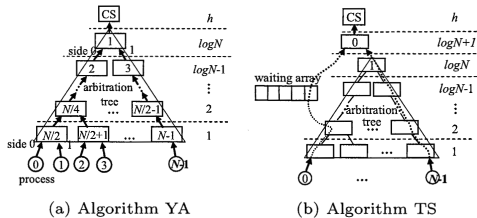
We evaluate ME algorithms with RMR complexity. The RMR complexity for ME algorithms is the number of remote memory references for each process during its entry and exit sections before and after each execution of its critical section respectively.

3 Previous Work

3.1 Yang's algorithm

Yang et al. [6] 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.



(a) Algorithm YA (b) Algorithm TS

Fig. 1 Algorithm YA and TS

4 Algorithm TS

4.1 Basic Ideas

In ME algorithms using an arbitration tree [4, 6, 3], a process that requires to visit all the nodes in the path from its leaf to the root.

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,

* 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
  $P[1..logN+1][0..N-1]$: $(\mathbf{0},\mathbf{1},\mathbf{2})$ *initially* 0
  $W[0..N-1]$: **integer** *initially* -1
  $Added[0..N-1]$: $(\mathbf{0},\mathbf{1},\mathbf{2})$ *initially* 0
  Add : **integer** *initially* $N-1$
  $Call$: **integer** *initially* $N-1$

Fig. 2 Shared Variables in TS

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.

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 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 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 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 an execution which is started at $l_1@p$(Func) and is completed at $l_2@p$(Func). In this section, "call" means that to execute 31@(TS). We omit proofs of the lemmas. The proofs are appeared in [5].

4.3.1 Exclusion

The following condition is satisfied for all the nodes from both sides in YA [6].

Condition 1 ([6]). At most one process concurrently executes [1@(Entry2PME(n, s)), 1@(Exit2PME(n, s))] for a pair of node n and side s.

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

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;
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

1: tmpadd := Add;
2: for h := skip down to 1 do
3:     node := ⌊(N+ID)/2^h⌋;
4:     rival := T[node];
5:     if rival ≠ ID then
6:         tmpadd := (tmpadd + 1) mod N;
7:         if Added[rival] = 0 then
8:             Added[rival] := 1;
9:             W[tmpadd] := rival;
10:        end if
11:    end if
12: end for
13: 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

Lemma 1. At most one process concurrently executes [18@(TS), 28@(TS)] for the pair of node 0 and side 0.

Theorem 1. TS ensures exclusion.

4.3.2 Starvation-freedom

We prove starvation-freedom for TS. All waiting loops in entry sections of TS are eventually finished iff TS ensures this property. For any process p, there are the following three waiting loops in its entry section.
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** $P[h][p] > 1$)
The following lemma hold for YA [6].

Lemma 2 ([6])**.** Wait 1 is eventually finished.

In TS, each process operates same as YA in the case that it and another process concurrently execute each entry section of 2PME at a same node. Thus, Lemma 2 holds in TS too.

Lemma 3. Wait 2 is eventually finished.

Lemma 4. Wait 3 is eventually finished.

By Lemma 2, 3 and 4, 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.

We first 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 was added to the array)
Update 2 at 31@(TS) (p was called)
The following lemma and corollary hold on the update. The proof is appeared in [5].

Lemma 5. If Update 2 is executed after any process p writes T to p, *Added*[p] is not updated until 8@p(TS) is completed.

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

We 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)), skipping nodes ([29@(TS), 31@(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. □

4.4.1 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 $O(\log N)$

4.4.2 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' 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), 12@(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

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

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.

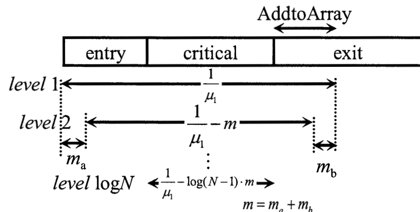


Fig. 8 The service at each level

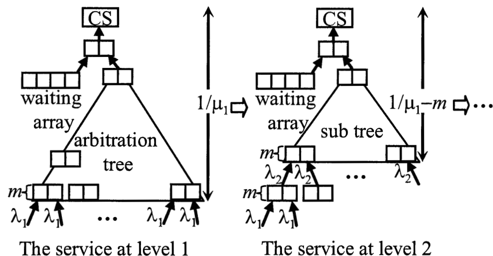
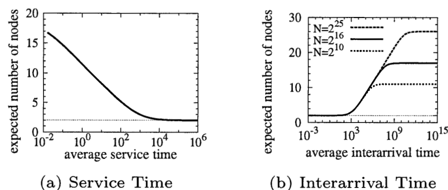


Fig. 9 The recursive calculus of λ_k and $\frac{1}{\mu_k}$

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

4.4.3 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 op-

Table 1　Case Study Prameters

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

eration are different for every processes and their average times among processes are varied in the range of $\pm 100\%$ of the values in Table (2). These execution times for each process is varied in the range of $\pm 50\%$ 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.

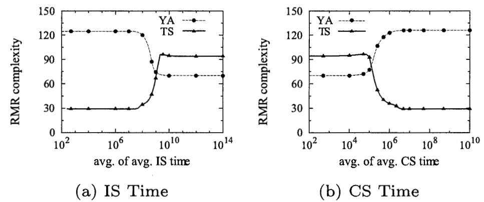
Table 2　Simulation Setup Prameters

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

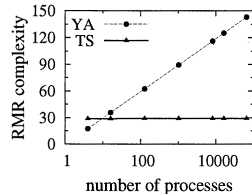


Fig. 12　RMR complexity and the number of processes

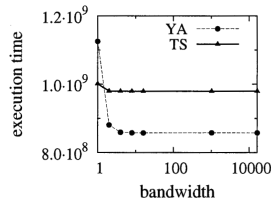


Fig. 13　Execution time and the bandwidth

5　Improving TS to FTS

In TS, each process has 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.

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 get 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 get 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 4 remote memory accesses at the beginning of its exit section before the next process starts its CS. The detail of FTS are described in [5].

We evaluate the execution time of FTS by simulation. Figure 15 shows the simulation results for the cases when (a) average CS time are different among processes, and (b) average CS time are common to processes. We find that the execution time is improved to almost the same value as YA.

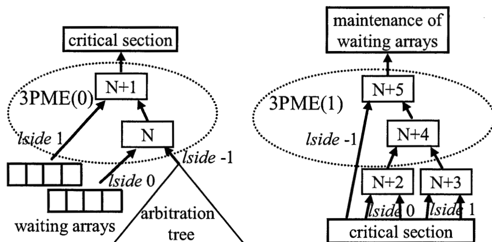
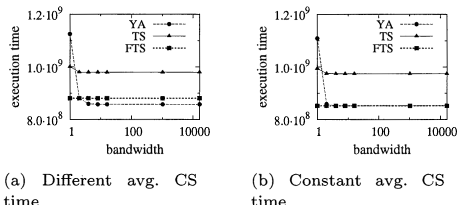


Fig. 14 Algorithm FTS



(a) Different avg. CS time

(b) Constant avg. CS time

Fig. 15 Execution time and the bandwidth

6 Conclusion

We proposed the mutual exclusion algorithm for distributed system with shared memory hierarchy. Though the RMR complexity of $O(\log N)$ is the same as the existing algorithm, 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.

Furthermore we improved the algorithm to reduce the actual execution time, and demonstrated the efficiency by simulation.

Though the proposed algorithms have the space 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] T. Suzuki, M. Inoue, and H. Fujiwara. Efficient Mutual Exclusion Algorithm for High System Congestion. *NAIST Information Science Technical Report, NAIST-IS-TR2009001*, 2009.

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