An Optimal Algorithm for Global Termination Detection in Shared-Memory Asynchronous Multiprocessor Systems

Ho-fung Leung and Hing-fung Ting, *Member, IEEE*

Abstract—In the literature, the problem of global termination detection in parallel systems is usually solved by message passing. In shared-memory systems, this problem can also be solved by using exclusively accessible variables with locking mechanisms. In this paper, we present an algorithm that solves the problem of global termination detection in shared-memory asynchronous multiprocessor systems without using locking. We assume a reasonable computation model in which concurrent reading does not require locking and concurrent writing different values without locking results in an arbitrary one of the values being actually written. For a system of n processors, the algorithm allocates a working space of $2n + 1$ bits. The worst case time complexity of the algorithm is $n + 2\sqrt{n} + 1$, which we prove is the lower bound under a reasonable model of computation.

Index Terms—Termination detect, shared-memory multiprocessor systems, optimality.

1 Introduction

Consider a system of asynchronous processors which communicate through shared-memory. In such a system, the processors cooperate to perform a task. A processor can send jobs to other processors at any time. Global termination of an execution refers to a situation in which all processors are either sleeping or about to sleep, and there is no job in the system. The detection of global termination of execution in such systems is nontrivial. Jobs could be sent when the termination detection algorithm is being executed. It is possible that global termination is wrongly reported if the algorithm is not designed to cater for all possible event sequences.

In the literature, global termination detection is achieved by message or token passing [1], [2], [3], [4], [5]. While these schemes are suitable for distributed systems, they incur overhead in shared-memory systems, in which message or token passing is not necessarily the most efficient means of information exchange among the processors. For shared-memory systems, a scheme for termination detection may depend on one or more exclusively accessible variables [6], [7], [8]. An example of such a scheme can be found in Crammond's paper [9], in which a garbage collection algorithm for an asynchronous multiprocessor system is described. In the system, each processor manages some private memory cells. A memory cell is considered to be garbage if it is not accessed by any processor. The garbage collection algorithm consists of two phases. In the first phase, nongarbage cells are marked. In the second phase, garbage cells are collected. Note that if a nongarbage cell contains a pointer, then the cell(s) pointed to should also be marked. If the pointer points to another processor's private memory, then that other processor should be informed to mark the cell(s) pointed to. Effectively, job is sent from a processor to another. The marking phase terminates when all the nongarbage cells are marked. Hence, a mechanism is needed to detect the completion of the first phase so that the second phase can start. To detect the termination of the marking phase, the algorithm explicitly makes use of a single global variable (called "a global counter of *indirect pointers to mark*"). This global counter is initialized to zero. Every time a processor sends job to another processor, this global counter is increased. Similarly, a processor decreases the value of the global counter when it finishes a job received from other processors. Termination is detected if every processor stops and the value of this global counter returns to zero. Obviously, frequent locking of this global counter is needed. This reduces the efficiency of the system as a processor needs to wait for the lock to be released.

In this paper, we present an algorithm for global termination detection in shared-memory asynchronous multiprocessor systems. An important feature of the algorithm is that it requires no locking at all. The memory model we use is a concurrent-read concurrent-write (CRCW) model. In other words, we make the following assumptions on the shared-memory architecture: The value of a variable can be read simultaneously by more than one processor without having to be locked first, and if more than one processor simultaneously write (possibly different) values to a variable, then the actual value written would be one arbitrarily chosen from these values. For a system of n processors, the algorithm needs $2n + 1$ bits as its working space. Global termination can be detected with a worst case time complexity of $n + 2\sqrt{n} + 1$. We note that a possible hardware solution to this problem is to make use of "composite registers" [10], [11].[1]

- *H. Leung is with the Department of Computer Science and Engineering, the Chinese University of Hong Kong, Shatin, Hong Kong. E-mail: lhf@cse.cuhk.edu.hk.*
- *H. Ting is with the Department of Computer Science, the University of Hong Kong, Pokfulam, Hong Kong. E-mail: hfting@cs.hku.hk.*

Manuscript received Nov. 23, 1994.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D95271.

This paper is organized as follows. In Section 2, we present a simple algorithm to illustrate the basic idea. The worst case time complexity of this simple algorithm is $2n + 2$. In Section 3, we present an improved algorithm, which has a worst case time complexity of $n + 2\sqrt{n} + 1$. We prove that this improved algorithm has achieved the lower bound in terms of time complexity under a reasonable model of computation. Section 4 concludes the paper.

2 The $\alpha\beta\gamma$ Algorithm

In this section, we describe a simple algorithm for solving the global termination detection problem. We are not concerned with how jobs are sent between processors. Instead, we assume that the following procedure and functions are provided by the system. We consider processors executing the **sleep** statement to be *sleeping* and they are called the *sleeping processors*. Processors which are not executing the **sleep** statement are said to be *awake* and they are called *awake processors*. However, it should be noted that we also regard a processor as sleeping even if it is awake but is going to sleep immediately without executing any other statements. Let $JQ[i]$ denote the job queue of processor P_i.

- **procedure** *enqueue*(J, $JQ[i]$): Adds the job J into $JQ[i]$. P_i will be awakened if it is sleeping. Control will not be returned to the caller until job J is successfully added to job queue $JQ[i]$.
- **function** *dequeue*($JQ[i]$): Removes and returns the first job from job queue $JQ[i]$. This function can be executed only by processor P_i.
- **function** *empty*($JQ[i]$): Returns *true* if job queue $JQ[i]$ is empty; *false* otherwise. This function can be executed simultaneously by more than one processor.

2.1 The Algorithm

For ease of presentation, we assume throughout the paper that there is a dedicated processor in the system, called the *detector*, which executes the termination detection algorithm. In an actual implementation, the role of such a detector can be played by any one of the processors in the system.

Following is a simple strategy for solving the global termination detection problem. The detector monitors all of the n processors as well as their job queues constantly. Once it finds that all processors are sleeping and all of their job queues are empty, it assumes that no new job would be generated and the system indeed globally terminates.

However, there is one problem with this strategy. It works only if the detector can simultaneously inspect all n processors and their job queues. However, the detector can only check one processor at a time. It cannot pronounce that the system globally terminates even if it finds out, at different time, that all of the n processors are sleeping and their job queues are empty. This is because a processor which is found to be sleeping could receive jobs from other processors immediately after it is checked. In order to arrive at a correct conclusion, the detector must also ensure that no job is sent during the time interval it checks these n processors.

The $\alpha\beta\gamma$ algorithm is an implementation of the above idea without using locking. The algorithm allocates $2n + 1$ bits in the shared memory, which are named the α_1-, α_2-, …, α_n-, β_1-, β_2-, …, β_n- ($1 \le i \le n$), and γ-bits. The α_i- and β_i -bits are set and reset by processor P_i only. The intuitive meaning of these bits are as follows. α_i is set if and only if there is a job sent to processor P_i. β_i is set if and only if processor P_i is awake. The γ-bit is set if and only if job has been sent by one processor to another since the last time γ is reset. The γ-bit can be set by any processor. However, it is reset by the detector only.

When the system needs to perform a task cooperatively, the detector adds initial jobs to the job queues of the processors, sets all α- and β-bits to 1 and the γ-bit to 0, and wakes up all the n processors. Then, every processor P_i, $1 \le i \le n$, executes the procedure *task*(i). Processor P_i sends a job J to another processor P_j by executing the procedure *sendJob*(J, j). The job J will then be added to the job queue of P_j. Note that procedure *sendJob*(J, j) is defined in such a way that it does not return until J is properly added to $JQ[j]$.

Once the system starts, the detector executes the procedure *monitor* which checks the system constantly. We say that the system globally terminates if and only if every processor either is executing the **sleep** statement in *task*(i), or has completely finished executing the statement immediately preceding the **sleep** statement. When the detector detects global termination, it adds a special job *FINISH* to the job queues of all the processors.

The procedures are defined as follows.

```
procedure task(i)
{
    forever do {
        if empty(JQ[i]) then {
            αi := 0;
            if empty(JQ[i]) then {
                βi := 0;
                sleep;
                /* until some other processor executes
                     enqueue(J′, JQ[i]) */
                βi := 1;
            }
            αi := 1;
        }
        J := dequeue(JQ[i]);
        if J = FINISH then
            exit
        else
            execute J;
    }
}
procedure sendJob(J, j)
{
    enqueue(J, JQ[j]);
    while (αj = 0 and ¬ empty(JQ[j]));
```

1. Composite register is an array-like hardware device that processes can write into their component in a composite register and they can read the entire register in an atomic step.

```
    /* At this point, either α_j = 1 or empty(JQ[j]) */
    γ := 1;
}
procedure monitor()
{
    h := 1;
    while (h = 1) {
        h := 0;
        /* now checks β_1, β_2, …, β_n in sequence */
        for i := 1, 2, …, n do h := h ∨ β_i;
        /* now checks γ */
        h := h ∨ γ;
        /* now resets γ */
        γ := 0;
    }
/* At this point, it can be sure that the system
globally terminates. */
    for i := 1, 2, …, n do enqueue(FINISH, JQ[i]);
}
```

2.2 The Correctness of the $\alpha\beta\gamma$ Algorithm

In this subsection, we show that the system, indeed, globally terminates when the procedure *monitor* executed by the detector exits the **while** loop and adds *FINISH* to the job queues of all the n processors.

Let t_i, $1 \le i \le n$, and t_γ denote, respectively, the time when the β_i - and γ-bits are checked by the detector for *the last time* before the procedure *monitor* terminates. From the definition of procedure *monitor*, it follows that $t_1 < t_2 < \cdots < t_n < t_\gamma$, β_i - bits are equal to 0 at t_i, $1 \le i \le n$, and γ-bit is equal to 0 at t_γ (Fig. 1).

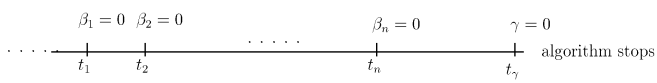


Fig. 1. The final round of execution.

In the following lemmas, J is a job generated by P_j. Let $t(J)$ denote the time when J is added to $JQ[i]$ for some i.

LEMMA 2.1. *If $t(J) < t_j$, then J will be removed from $JQ[i]$ before t_i.*

PROOF. Assume that $t(J) < t_j$. To add J to $JQ[i]$, P_j executes the procedure *sendJob*(J, i). Let t be the time when P_j sets the γ-bit in *sendJob*(J, i). We claim that $t < t_i$. If $t \ge t_i$, then the is γ-bit is set after t_i and it implies that $\gamma = 1$ at t_γ. This leads to contradiction.

Note that P_j will not set the γ-bit until $\alpha_i = 1$ or $JQ[i]$ is empty. If $JQ[i]$ is empty at t, then J must have been removed from $JQ[i]$ before $t(< t_i)$ and the lemma follows immediately. Otherwise, since $\alpha_i = 1$ implies $\beta_i = 1$, we have α_i and β_i are still set some time between $t(J)$ and $t(< t_i)$. Since $\beta_i = 0$ at t_i, P_i must have reset α_i, checked $JQ[i]$ and dequeued J before t_i. □

LEMMA 2.2. *For any $1 \le i \le j \le n$, P_i is sleeping and $JQ[i]$ is empty during the time interval t_i and t_j.*

PROOF. We prove it by induction on j. From the previous lemma and by the definition of t_1, it is trivial that the lemma is true for $j = 1$. Assume that the lemma is true for $j = k < n$ and consider the case when $j = k + 1$. From the induction hypothesis, we have, for all $1 \le i \le k$, P_i is sleeping and $JQ[i]$ is empty between time t_i and t_k. All of these processors remain sleeping in the time interval $[t_k, t_{k+1}]$ unless there is some awake processor adding job to their job queues. However, for any awake processor P_l, we must have $l > k$ and from the previous lemma, any job sent to P_i by P_l at or before $t_{k+1} \le t_l$ is removed from $JQ[i]$ before $t_i \le t_k$. Hence, all of these sleeping processors remain sleeping and their job queues remain empty between time t_k and t_{k+1}.

We claim that $JQ[k + 1]$ is also empty at t_{k+1}. For any job J in $JQ[k + 1]$ added by P_l at or before t_{k+1}, there are two possible cases

1) $l > k + 1$. As $t(J) < t_l$, from the previous lemma, we have J removed from $JQ[k + 1]$ before t_{k+1}. In other words, at t_{k+1}, there is no job in $JQ[k + 1]$ which is added by P_l with $l > k + 1$.
2) $l < k + 1$. Lemma 2.1 asserts that all the jobs added by P_l before t_l are removed from $JQ[k + 1]$ before t_{k+1}. From the previous discussion, we know that P_l is asleep during the time interval $[t_l, t_{k+1}]$ and there is no job J from P_l with $t_l \le t(J) \le t_{k+1}$. Hence, at t_{k+1}, there is no job in $JQ[k + 1]$ which is added by P_l with $l < k + 1$.

Together with the fact that $\beta_{k+1} = 0$ at t_{k+1}, we have P_{k+1} is sleeping and $JQ[k + 1]$ is empty at t_{k+1}. Therefore, the lemma is true for $j = k + 1$ and by mathematical induction, the lemma is proved for all $1 \le i \le j \le n$. □

An immediate consequence of the previous lemma is that all the P_is are sleeping and all the $JQ[i]$s are empty at t_n. Hence, no more jobs will be generated after t_n and we have the following theorem:

THEOREM 2.1. *The system globally terminates at time t_n.*

2.3 The Complexity of the Algorithm

First of all, let us define the measure of complexity in terms of the number of queries[2] made during the execution. For any query $\mathcal{Q}$, we say $\mathcal{Q}$ is *free* if it is made before the system globally terminates. Otherwise, we say that $\mathcal{Q}$ is *expensive*. The number of free queries is not our concern here. In fact, it is unbounded by any function of n, the number of processors in

the system. On the other hand, the number of expensive queries depends on the algorithm. It is essential to minimize the number of expensive queries so that the detector would detect global termination as soon as possible. The *worst case time complexity* for any global termination detection algorithm is defined to be the maximum, over all possible executions, number of expensive queries that the algorithm makes.

THEOREM 2.2. *The worst case time complexity of the $\alpha\beta\gamma$ algorithm is* $2n + 2$.

PROOF. When the system globally terminates, all the β-bits are reset. However, the γ-bit may still be equal to 1. It takes at most n expensive queries o check the β-bits before the γ-bit is checked and reset by the detector. As the γ-bit is equal to 1 when it is checked, the detector makes another $n + 1$ queries before it can be sure that the system globally terminates. Hence, the worst case time complexity of the $\alpha\beta\gamma$ algorithm is $2n + 2$. □

3 An Optimal Algorithm for Detecting Global Termination

The $\alpha\beta\gamma$ algorithm has a worst case time complexity of $2n + 2$ because it wastes a lot of expensive queries. Even though the system has globally terminated, the n β-bits are checked all over again if the γ-bit is found to be 1. Intuitively, one can reduce the waste and design a more efficient algorithm by checking the γ-bit more frequently. In the following subsection, we present an improved algorithm based on this idea. We derive a lower bound on the time complexity for solving the problem under a reasonable model of computation and the algorithm is optimal under this model.

3.1 An Improved Algorithm

The improved algorithm is very similar to the $\alpha\beta\gamma$ algorithm except that the γ-bit is checked whenever the detector has checked $\sqrt{n}$ β-bits. The procedures for checking and updating the α- and β-bits are exactly the same as those in the $\alpha\beta\gamma$ algorithm. In other words, the procedures *task* and *sendJob* are unchanged. However, the procedure *monitor* for detecting global termination is modified as follows.

```
procedure new_monitor()
{
    h := 1;
    while (h = 1) {
        h := 0; i := 1;
        while (i ≤ n) {
            h := β_i ∨ h;
            i := i + 1;
            if i mod √n = 0 then {
                h := h ∨ γ ;
                if h = 1 then {h := 0; γ := 0; i := 1};
            }
        }
        h := h ∨ γ ;
        γ := 0;
    }
/* At this point, it can be sure that the system
globally teminates. */
    for i := 1, 2, …, n do enqueu(FINISH, JQ[i]);
}
```

The correctness of the improved algorithm follows directly from Theorem 2.1. To find the number of expensive queries made in the worst case, we observe that when the system globally terminates, the γ-bit will be checked and reset after at most $\sqrt{n}$ expensive β-queries. Then, all the n β-bits and the γ-bit will be equal to 0. Then, the detector needs to check all the β-bits all over again and all these n queries are expensive. In addition, it still needs to check the γ-bit whenever it has checked $\sqrt{n}$ β-bits, there are $\sqrt{n}$ expensive queries for checking the γ-bit. Adding the number of these queries together, we have the following theorem:

THEOREM 3.1. *The worst case time complexity of the improved algorithm is* $n + 2\sqrt{n} + 1$.

3.2 A Lower Bound

Any algorithm which solves the global termination detection problem has to check whether a processor is sleeping and whether there is job sent in a particular time interval. To abstract all the implementation details, we assume that there are two kinds of queries in our model of computation:

1) *β-query.* When the detector makes a *β-query* $\mathcal{Q}_\beta(i)$, the system responds with a "1" if processor P_i has not finished all the jobs on hand. Otherwise, it responds with a "0."

2) *γ-query*. When the detector makes a γ-query $\mathcal{Q}_\gamma$ at time t, the system responds with a "1" if there is a new job generated during the time interval $[t_l, t]$, where t_l is the time when the previous γ-query was made.[3] Otherwise, it responds with a"0."

In this subsection, we show that any global termination detection algorithm has to made at least $n + 2\sqrt{n} + 1$ expensive queries in the worst case to arrive at a correct conclusion. This shows that the improved algorithm is optimal under this model of computation.

We derive a lower bound on the worst case time complexity for solving the problem by adversary arguments (see the book by Horowitz and Sahni [12]). First, we construct an oracle $\mathcal{F}$ which answers all the queries made by the detector. Then, we show that there exists a scenario in which the system responds to the queries in exactly the same way as the oracle does. Finally, we prove that answering these queries in such a way would force any global termination detection algorithm to make at least $n + 2\sqrt{n} + 1$ expensive queries.

First of all, let us have some definitions. For any query $\mathcal{Q}$, we say $\mathcal{Q}$ is *active* if $\mathcal{F}$ responds with a "1" to the query. Oth-

2. i.e., the checking of β_i -bits and the γ-bit.

3. If the detector has not made any γ-query before t, then t_l is the time when the algorithm starts.

4. B_1 is defined to be the set of β-queries made before the detector makes the first γ-query.

erwise, we say it is *inactive*. For any $i \geq 1$, let $\mathcal{Q}_{\gamma,i}$ be the ith γ-query made by the detector and B_i be the set of all β-queries made between $\mathcal{Q}_{\gamma,i-1}$ and $\mathcal{Q}_{\gamma,i}$.[4] Let $t_{i,1}$ and $t_{i,2}$ be the time when the detector makes the first and the last β-query in B_i, respectively. We say B_i is *active* if $\mathcal{Q}_{\gamma,i}$ is an active query. Otherwise, B_i is *inactive* (Fig. 2).

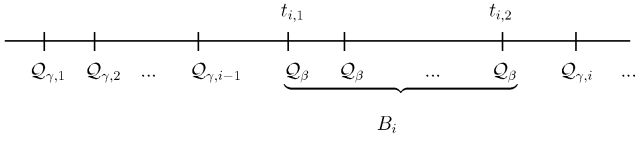


Fig. 2. The definition of B_i.

The oracle $\mathcal{F}$ responds to the queries as follows.

1) All the β-queries are inactive, i.e., $\mathcal{F}$ always responds with a "0" to any β-query.
2) $\mathcal{Q}_{\gamma,1}$ is an active query. For any query $\mathcal{Q}_{\gamma,i}$, $i > 1$, response of $\mathcal{F}$ depends on the size of B_j, B_{j+1}, …, B_{i-1} where j is the largest integer such that $j < i$ and B_j is active. More precisely, $\mathcal{F}$ responds with a "1" to $\mathcal{Q}_{\gamma,i}$ if and only if
   - $|B_j| < |B_i|$, and
   - $|B_i| < n + 2\sqrt{n}$, and
   - $\sum_{j<k<i} |B_k| < n$.

We now describe a scenario such that the system will respond to the queries exactly the same as $\mathcal{F}$ does. For any query $\mathcal{Q}_\beta(j)$ made at time t, if processor P_j is still awake just before t, it finishes all the jobs on hand and goes to sleep before t. This is possible because of the asynchronous nature of the problem. For query $\mathcal{Q}_{\gamma,i}$, there is no problem if it is inactive. The processors simply do not generate any job in the time interval the detector makes the queries $\mathcal{Q}_{\gamma,i-1}$ and $\mathcal{Q}_{\gamma,i}$. However, if $\mathcal{Q}_{\gamma,i}$ is active, then we must be sure that there is at least one processor awake just before $t_{i,1}$, the time when the first β-query in B_i is made. Otherwise, no new job can be generated and the response to $\mathcal{Q}_{\gamma,i}$ cannot possibly be "1." Intuitively, any awake processor should constantly send jobs to all the other processors whenever it is possible, i.e., not in a time interval corresponding to an inactive B_i.[5] Assume that the system behaves in this way. The following lemma shows that there is always an awake processor just before $t_{i,1}$.

LEMMA 3.1. *$\mathcal{Q}_{\gamma,i}$ is an active query, then there is at least one awake processor just before $t_{i,1}$, the time when the first β-query in B_i is made.*

PROOF. We prove the lemma by induction. Assume that there are totally m active γ-queries and let $\mathcal{Q}_{\gamma,i_j}$ be the jth active γ-queries. We prove the lemma by induction on j. From the construction of $\mathcal{F}$, we have $i_1 = 1$ and the lemma is true because the processors are awakened before the detector makes any queries. Assume that the lemma is true for all j with $1 \leq j < k \leq m$ and consider the query $\mathcal{Q}_{\gamma,i_k}$. Let $i_{k-1} = s$ and $i_k = t$ (Fig. 3).

By mathematical induction, there is at least one awake processor just before $t_{s,1}$, the time when the first β-query in B_s is made. By the nature of the system, this processor will wake up all the other $n-1$ processors and, since the system is asynchronous, we can assume that it is done before $t_{s,1}$. After that, there may be processors which are forced to sleep because of the β-queries. However, the sleeping processors could be awakened by other processors immediately after the β-queries are made. Hence, we can assume that all the processors are awake immediately after $t_{s,2}$. By the construction of $\mathcal{F}$, there are at most $n-1$ β-queries being made between $t_{s,2}$ and $t_{t,1}$. Therefore, at most $n-1$ processors would be forced to sleep and there is at least one processor awake just before $t_{t,1}$. Hence, it is also true for $j = k$ and, by mathematical induction, the lemma is true for all $1 \leq j \leq m$. □

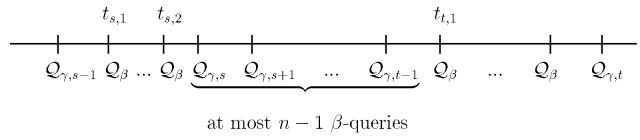


Fig. 3. The β and γ queries.

THEOREM 3.2. *Any global termination detection algorithm has to make at least $n + 2\sqrt{n} + 1$ expensive queries.*

PROOF. Without loss of generality, we assume that there is no inactive B_i with size greater than $n + 2\sqrt{n}$. Otherwise, the theorem is obviously true. Let $\mathcal{Q}_{\gamma,l}$ be the last active γ-query.[6] From the proof of the previous lemma, there is a scenario in which all the n processors are awake when an active γ-query is made. Hence, there must be at least n β-queries after $\mathcal{Q}_{\gamma,l}$. Otherwise, we cannot be sure that the system has globally terminated. In other words, there must be a $j(> l)$ such that B_l is active, B_{l+1}, …, B_j are inactive and $\sum_{l<k\leq j} |B_k| \geq n$. We can assume that the system globally terminates before $t_{l+1,1}$, the time when the first β-query in B_{l+1} is made, and thus all the

5. Of course, any awake processor should go to sleep immediately before any β-query is made about it.

6. There is at least one active γ-query, $\mathcal{Q}_{\gamma,1}$.

queries made after $t_{l+1,1}$ are expensive. From the construction of $\mathcal{F}$, we have $|B_l| \geq |B_k|, l < k \leq j$. Hence, there are at least $j - l > \frac{n}{|B_i|}$ expensive γ-queries. Totally, there are at least $n + \frac{n}{|B_i|}$ expensive queries being made after $t_{l+1,1}$. However, observe that the behavior of the system can be changed a little so that it will answer the queries exactly the same way as before but there are even more expensive queries. In the new scenario, the system globally terminates immediately before $t_{l,1}$, the time when the first β-query in B_l is made. As B_l is active, from the previous lemma, there is still an awake processor and a new job is generated before $t_{l,1}$. So, even if the system globally terminates just before $t_{l,1}$, $\mathcal{Q}_{\gamma l}$ is still active. In the new scenario, the total number of expensive queries being made would be at least $n + |B_l| + 1 + \frac{n}{|B_i|}$. Minimize the expression and we have the number of expensive queries being made are greater than $n + 2\sqrt{n} + 1$. □

4 Conclusions

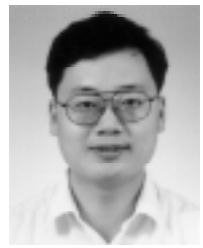
In this paper, we present the $\alpha\beta\gamma$ algorithm as well as an improved version for global termination detection in a shared-memory asynchronous multiprocessor system. Unlike those used in the literature, our algorithms require neither message passing nor locking of global variables. We show that the worst case time complexity of the improved version is $n + 2\sqrt{n} + 1$. Using the technique of oracle construction, we proved that this is the lower bound under a reasonable model of computation.

Acknowledgment

We thank the anonymous referees for their invaluable comments.

References

[1] M. Raynal, *Distributed Algorithms and Protocol.* Chichester: Wiley, 1988.
[2] E.W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren, "Derivation of a Termination Detection Algorithm for a Distributed Computation," *Information Processing Letters*, vol. 16, no. 5, pp. 217-219, 1983.
[3] F. Mattern, "An Efficient Distributed Termination Test," *Information Processing Letters*, vol. 31, no. 4, pp. 203-208, May 1989.
[4] S. Chandrasekaran and S. Venkatesan, "A Message-Optimal Algorithm for Distributed Termination Detection," *J. Parallel and Distributed Computing*, vol. 8, no. 3, pp. 245-252, Mar. 1990.
[5] T.-H. Lai, Y.-C. Tseng, and X. Dong, "A More Efficient Message-Optimal Algorithm for Distributed Termination Detection," *Proc. Sixth Int'l Parallel Processing Symp.*, V.K. Prasanna and L.H. Canter, eds., Beverly Hills, Calif., pp. 646-649, IEEE CS Press, Mar. 1992.
[6] L. Lamport, "On Interprocess Communication: Part I: Basic Formalism," *Distributed Computing*, vol. 1, no. 2, pp. 77-85, 1986.
[7] L. Lamport, "On Interprocess Communication: Part II: Algorithms," *Distributed Computing*, vol. 1, no. 2, pp. 86-101, 1986.
[8] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
[9] J.A. Crammond, "A Garbage Collection Algorithm for Shared Memory Parallel Processors," *Int'l J. Parallel Programming*, pp. 497-522, 1990.
[10] J.H. Anderson, "Composite Registers," *Proc. Ninth ACM Symp. Principles of Distributed Computing*, pp. 15-30, 1990.
[11] Y. Afek, H. Attiya, D. Dolev, E. Gafni, and M. Merritt, "Atomic Snapshots of Shared Memory," *Proc. Ninth ACM Symp. Principles of Distributed Computing*, pp. 1-14, 1990.
[12] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.



Ho-fung Leung received his BSc and MPhil degrees in computer science from the Chinese University of Hong Kong in 1985 and 1988, respectively. He did his PhD at Imperial College of Science, Technology, and Medicine and received his PhD degree in computing from the University of London in 1992. He is currently an associate professor in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His major research interests include constraint satisfaction problems, fuzzy set theory, logic programming, and their implementation and applications.

Hing-fung Ting received his BSc in computer science from the Chinese University of Hong Kong in 1985 and his PhD in computer science from Princeton University in 1992. He is currently an assistant professor in the Department of Computer Science at the University of Hong Kong. His major research interests include computational biology and approximation algorithms.