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

Ho-Fung Leung Hing-Fung Ting
Department of Computer Science
The Chinese University of Hong Kong
Shatin, Hong Kong
Email: lhf@cs.cuhk.hk; hfting@cs.cuhk.hk

Abstract

In the literature the problem of global termination detection in shared-memory asynchronous multiprocessor systems is solved by using exclusively accessible variables and locking mechanism. In this paper we present an algorithm that solves this problem without using locking. We assume that concurrent reading does not require locking and concurrent writing different values without locking results in 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.*

1 Introduction

Consider a system of shared-memory asynchronous autonomous processors in which a processor can *'throw'* jobs to other processors at any time. The detection of global termination of execution in such a system is nontrivial: jobs could be thrown when the termination detection algorithm is being executed, and it is possible that global termination is wrongly reported if a processor that has finished its execution receives jobs from other processors. Traditionally, a detection algorithm would depend on one or more variables that can only be accessed exclusively during normal execution of the underlying computation process, and locking is inevitable in updating these variables [1]. This would affect the efficiency of the system as a processor needs to wait for the lock to be released.

An example of this can be found in [2], in which a garbage collection algorithm for an asynchronous multiprocessor system is described. In the garbage collection algorithm one processor can trace a pointer and tell another processor to mark the useful memory cells the latter manages. Effectively the former processor throws jobs to the latter. The marking phase terminates when all processors have marked the useful memory cells they manage and none of them is going to give jobs to any other processors. To detect the termination of the marking phase, the algorithm explicitly makes use of a global variable (called 'a global counter of *indirect pointers to mark*'). This global counter is initialized to zero. Every time a processor throws work to another processor, this global counter is increased by the amount of work thrown. Similarly, a processor decreases the value of the global counter by the amount of work finished if the work was 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 during the execution of the underlying computation.

In this paper we present an algorithm that 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. Termination can be detected with a worst case time complexity of $n + 2\sqrt{n} + 1$.

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 shall prove that this improved algorithm has achieved the lower bound in terms of time complexity under a very general 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 the details how jobs are thrown between processors. Instead, we assume that the following procedures are provided by the system:

- $enqueue(J, JQ[i])$: adds the job J into $JQ[i]$, the job queue of processor P_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 $JQ[i]$.
- $dequeue(JQ[i])$: removes the first job from job queue $JQ[i]$.
- $empty(JQ[i])$: returns 1 if job queue $JQ[i]$ is empty; 0 otherwise.

2.1 The Algorithm

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

Following is a simple strategy for solving the global termination detection problem. The scheduler 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 can be sure 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 scheduler can simultaneously inspect all n processors and their job queues. This requirement cannot be achieved in a system of shared-memory asynchronous autonomous processors. Since the scheduler 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: a processor which is found to be sleeping could receive jobs from other processors soon after it is checked. In order to arrive at a correct conclusion, the scheduler must also be sure that no job is generated during the time interval it checks the 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, namely the α_1-, α_2-, ..., α_n-, β_1-, β_2-, ..., β_n- $(1 \leq i \leq n)$ and γ-bits. The α_i- and β_i-bits are set/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 job thrown to processor P_i. β_i is set if and only if processor P_i is busy. The γ-bit is set if and only if job has been thrown by one processor to another since the last time γ is reset. The γ-bit is set by any job-throwing processor. However, it is reset by the scheduler only.

Initially all of the $2n+1$ bits are reset to 0. When the system needs to perform a task cooperatively, the scheduler adds *start-up* jobs to the job queues of the processors and wakes up all the n processors. Then, every processor P_i, $1 \leq i \leq n$, executes the procedure $task(i)$. Whenever a processor P_i throws a job J to another processor P_j, it executes the procedure $request(J, j)$. The job J will then be added to the job queue of P_j. Note that procedure $request(J, j)$ is defined in such a way that control will not be returned to P_i until P_j is aware that a new job J is added to its job queue.

Once the system starts, the scheduler executes the procedure *monitor* which checks the system constantly. When the system globally terminates, the scheduler adds a special job *FINISH* into the job queues of all the processors.

The procedures are defined as follows.

```
task(i)
{
  β_i := 1;
  α_i := 1;
  forever do {
          if (empty(JQ[i])) {
            α_i := 0;
            if (empty(JQ[i])) then {
              β_i := 0;
              sleep; /* until some other processor
                     executes enqueue(J', JQ[i]) */
              β_i := 1;
            }
            α_i := 1;
          }
          J := dequeue(JQ[i]);
          if J = FINISH then exit
          else execute J;
  }
}

request(J, j)
{
  enqueue(J, JQ[j]);
  while ( α_j = 0 );
  /* now sets γ */
  γ := 1;
}

monitor()
{
  r := 1;
  while (r = 1) {
        r := 0;
        /* now checks β_1, β_2, ..., β_n in sequence */
        for i := 1, 2, ..., n do r := r ∨ β_i;
        /* now checks γ */
        r := r ∨ 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 scheduler 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 the time when the β_i- and γ-bits are checked by the scheduler for *the last time* before the procedure *monitor* terminates.

From the definition of procedure *monitor*, it follows that $t_1 < t_2 < ... < t_n < t_\gamma$ and the β_i-, $1 \le i \le n$, and γ-bits are equal to 0 at t_i and t_γ respectively(figure 1). The t_i's are important to our analysis.

Let J be a job generated by P_j and $t(J)$ be the time when it 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 has to execute the procedure $request(J, i)$. Let t be the time when P_j sets the γ-bit in $request(J, i)$. We claim that $t < t_i$. If $t_j < t_i$, then this claim is obviously true. Otherwise, for the case $t_i < t_j$, if $t \ge t_i$, then the γ-bit is set between t_i and t_j. However, it implies that $\gamma = 1$ at t_γ and this leads to contradiction.

Observe that P_j will not set the γ-bit until $\alpha_i = 1$ and $\alpha_i = 1$ 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 $1 \leq i \leq j \leq n$, 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 let us consider the case when $j = k+1$. From the induction hypothesis, we have, for all $1 \leq i \leq k$, 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, any awake processor P_l must have $l > k$ and from the previous lemma, any job sent to P_i by P_l at or before $t_{k+1} \leq t_l$ is removed from $JQ[i]$ before $t_i \leq t_k$. 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 $t_l \leq t(J) \leq t_{k+1}$. 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 $1 \leq i \leq j \leq n$. □

An immediate consequence of the previous lemma is all the P_i are sleeping and all the $JQ[i]$ are empty at t_n. Hence, no more job 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

In this subsection, we analyze the complexity of the $\alpha\beta\gamma$ algorithm. First of all, let us define the measure of complexity in terms of the number of queries† 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 for the algorithm to minimize the number of expensive queries so that the scheduler would detect global termination as soon as possible. Hence, 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 be still equal to 1. It takes at most n expensive queries to check the β-bits before the γ-bit is checked and reset by the scheduler. As the γ-bit is equal to 1 when it is checked, the scheduler 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 often. In the following subsection, we present an improved algorithm based on this idea. We then derive a lower bound on the time complexity for solving the problem under a very general model of computation. This lower bound shows that the improved algorithm is indeed optimal.

*We regard P_i to be sleeping even if it is awake but is going to sleep immediately without executing any other instruction.
† *i.e.*, the checking of β_i-bits and the γ-bit.

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 scheduler has checked $\sqrt{n}$ β-bits. The procedure for checking and updating the α- and β-bits are exactly the same as in the $\alpha\beta\gamma$ algorithm. However, the procedure for detecting global termination is modified as follows. As in the $\alpha\beta\gamma$ algorithm, the improved algorithm checks the β_1-, β_2-, ..., β_n-bits and γ-bit in sequence and announces global termination if all the $n+1$ bits are found to be 0. However, after checking every $\sqrt{n}$ β-bits and finding out they are equal to 0, it checks the γ-bit. If $\gamma = 0$, then the scheduler proceeds to check the next $\sqrt{n}$ β-bits. Otherwise, it resets the γ-bit and starts checking all the $n+1$ bits again. Effectively, only the procedure *monitor* needs to be modified.

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 queries. Then, all the $n+1$ bits will be equal to 0. As the scheduler still needs to check the γ-bit whenever it has checked $\sqrt{n}$ β-bits, there are $\frac{n}{\sqrt{n}}$ expensive queries for checking the γ-bit. Adding all these expensive 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

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

To abstract all the implementation details, we assume that there are two kinds of queries in our model of computation:

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

γ-query When the scheduler makes a γ-query $\mathcal{Q}_\gamma$ at time t, the system responds with an '1' if there is new job generated during the time interval $[t_l, t]$, where t_l is the time when the previous γ query was made[‡]. Otherwise, it responds with a '0'.

We derive a lower bound on the worst case time complexity for solving the problem by adversary arguments (see [3]). Firstly, we construct an oracle $\mathcal{F}$ which answers all the queries made by the scheduler. Then, we show that there exist a scenario in which the system responds the queries exactly the same way as the oracle does. Finally, we prove that answering the queries in such 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 an '1' to the query. Otherwise, we say it is *inactive*. For any $i \geq 1$, let $\mathcal{Q}_{\gamma,i}$ be the ith γ-query made by the scheduler and B_i be the set of all β-queries made between $\mathcal{Q}_{\gamma,i-1}$ and $\mathcal{Q}_{\gamma,i}$ [§]. Denote t_{i1} and t_{i2} to be the time when the scheduler 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* (Figure 2).

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 $B_j, B_{j+1}, \ldots, B_{i-1}$ where j is the largest integer such that $j < i$ and B_j is active. More precisely, $\mathcal{F}$ responds with an 1 to $\mathcal{Q}_{\gamma,i}$ if and only if
   - $|B_j| < |B_i|$, and
   - $|B_i| < n + 2\sqrt{n}$, and

[‡]If the scheduler has not made any γ-query before t, then t_l is the time when the algorithm starts.

[§]B_1 is defined to be the set of β queries made before the scheduler makes the first γ-query.

- $\sum_{j<k<i} |B_k| < n$.

Obviously, there are at most $n+1$ active γ-queries.

Now, let us 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 scheduler 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_{i1}, 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 'throw' jobs to all the other processors whenever it is possible, *i.e.*, not in a time interval corresponding to an inactive B_i[¶]. Assume that the system behaves in this way. The following lemma shows that there is always an awake processor just before t_{i1}.

Lemma 3.1 If $\mathcal{Q}_{\gamma,i}$ is an active query, then there is at least one awake processor just before t_{i1}, the time when the first β query in B_i is made.

Proof We prove the lemma by induction. Assume that there are totally $m \le n+1$ active γ-queries and let $\mathcal{Q}_{\gamma,i_1}, \mathcal{Q}_{\gamma,i_2}, \ldots, \mathcal{Q}_{\gamma,i_m}$ be the first, second, ..., mth active queries. From the construction of $\mathcal{F}$, we have $i_1 = 1$ and the lemma is true because the processors are awakened before the scheduler making any queries. Assume that the lemma is true for all i_j with $1 \le j < k \le m$ and let us consider the query $\mathcal{Q}_{\gamma,i_k}$. Let $i_{k-1} = s$ and $i_k = t$ (Figure 3).

By mathematical induction, there is at least one awake processor just before t_{s1}, 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_{s1}. 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_{s2}. By the construction of $\mathcal{F}$, there is at most $n-1$ β-queries being made between t_{s2} and t_{t1}. Therefore, at most $n-1$ processors would be forced to sleep and there is at least one processor awake just before t_{t1}. Hence, it is also true for $j=k$ and by mathematical induction, the lemma is true for all $1 \le j \le m$. □

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 $\mathcal{Q}_{\gamma,i}$ be the last active γ-query[‖]. From the proof of 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 $\mathcal{Q}_{\gamma,i}$. Otherwise, it cannot be sure the system has globally terminated. In other words, there must be a j such that B_i is active, $B_{i+1}, \ldots, B_j$ are inactive and $\sum_{i<k\le j} |B_k| \ge n$. We can assume the system globally terminates before t, the time when first β-query in B_{i+1} is made, as all the queries made after t are inactive anyway. In other words, there is an scenario in which all the queries made after t are expensive. From the construction of $\mathcal{F}$, we have $|B_i| \ge |B_k|$, $i < k \le j$. Hence, there are at least $j - i > \frac{n}{|B_i|}$ expensive γ-queries. Totally, there are at least $n + \frac{n}{|B_i|}$ expensive queries being made after t. 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', the time when the first β-query in B_i is made. As B_i is active, from the previous lemma, there is still awake processor and new job is generated before t'. So, even if the system globally terminates just before t', $\mathcal{Q}_{\gamma,i}$ is still active. In the new scenario, the total number of expensive queries being made would be $n + |B_i| + 1 + \frac{n}{|B_i|}$. Minimize the expression and we have the number of expensive queries being made are greater that $n+2\sqrt{n}+1$.
□

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

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

4 Conclusions

In this paper we presented 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, these algorithms require no locking for detection of global termination. We showed that the the worst 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 when the queries are restricted to the β- and γ- query. It should be obvious that our lower bound proof can be generalized to show that it is still the lower bound if we allow the more general type of γ-query $\mathcal{Q}_\gamma(t_i, t_j)$, which returns an '1' if there is new job generated during the time interval $[t_i, t_j]$ for arbitrary t_i and t_j. It is still an open problem whether the same lower bound holds for other type of queries.

References

[1] K.M. Chandy and J. Misra, *Parallel Program Design: a foundation* (Addison-Wesley, 1988).

[2] J.A. Crammond, A Garbage Collection Algorithm for Shared Memory Parallel Processors, *International Journal of Parallel Programming* **17** (1990) 497-522.

[3] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms* (Computer Science Press, 1978).

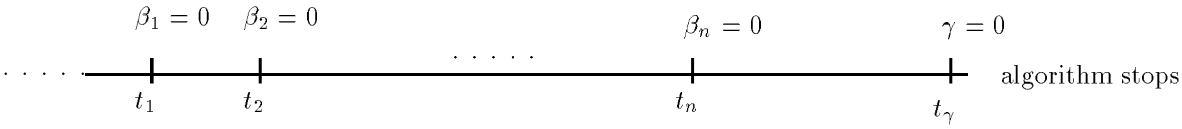

Figure 1

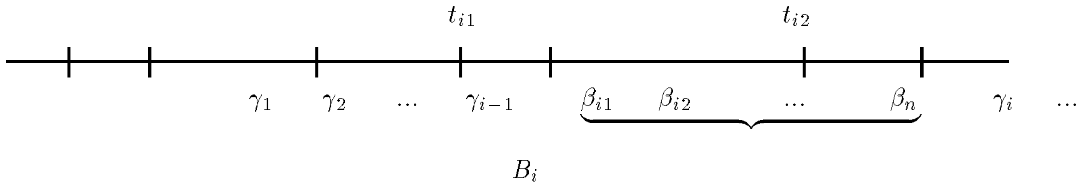

Figure 2

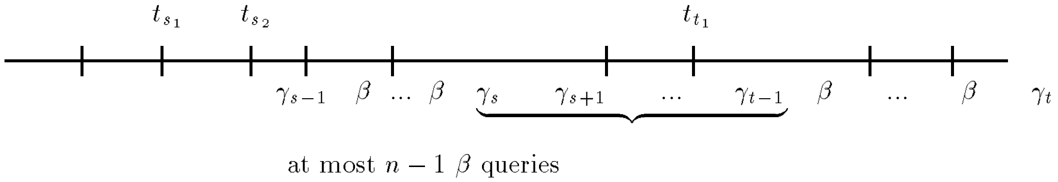

Figure 3