UNIVERSITY OF CALGARY

Title of Thesis

Second ThesisTitle Line

by

Student's name

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF NAME OF DEGREE IN FULL

DEPARTMENT OF NAME OF DEPARTMENT

CALGARY, ALBERTA

monthname, year

Abstract

Acknowledgements

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
U of C	University of Calgary

Chapter 1

Introduction

In this chapter, we will introduce the task allocation problem as well as its dynamic version and survey the past work related to it.

Task allocation problem, also called do-all problem, is already a foundational topic in distributing computing. During the last two decades, significant research was dedicated to studying the task allocation problem in various models of computation, including message passing, shared memory, etc. under specific assumption about the asynchrony and failures.

In this thesis, we consider the dynamic version of the task allocation problem via randomized asynchronous shared memory. The dynamic version could be described as follows:

p processes cooperatively perform a set of tasks in the presence of adversity and the tasks are injected dynamically by the adversary during the execution.

1.1 Related Work

1.2 Statement of Results

Chapter 2

The Computational Model and Problem Statement

In this chapter, we will formally describe the system model we are working in and introduce our dynamic task allocation object.

2.1 Computational Model

The computational model we consider is the standard asynchronous shared memory model with n processes. Every process executes its program by taking steps and up to $n-1$ processes may fail by crashing. A step is defined to be the execution of all local computations followed by an operation on a shared object. In this thesis, we consider the system that supports atomic compare-and-swap (CAS) object.

A CAS object v is a shared memory object stores a value from some set and supports two atomic synchronization operations $v.read()$ and $v.CAS((x,y),(a,b))$. Operation $v.read()$ returns the current value of the object and leaves the value unchanged. A $v.CAS((x,y),(a,b))$ takes the memory locations and writes new value into it only if it matches pre-supplied "expected" values (x,y), i.e, if current value of v are exactly (x,y), then the operation $v.CAS((x,y),(a,b))$ succeeds, and the value of v is changed to be (a,b) and $true$ is returned. Otherwise, the operation fails, and the current value of v remains unchanged and $false$ is returned.

In addition, each process has a unique identifier i which is from an unbounded namespace. A process can execute local coin flip operations that returns an integer value distributed uniformly at random from an arbitrary finite set of integers. In the following discussion, we use $random(s)$ to return a value distributed uniformly at random from the set $\{0,1,2,...,s-1\}$. We analyze our algorithm under the assumption of a strong adaptive adversary. At any

point of time, it can see the entire history and know the states of all processes. Base on such history, it decides which process takes the next step.

2.2 Problem Statement

Dynamic task allocation is the problem in which n processes perform a set of tasks cooperatively and the tasks are inserted dynamically by the adversary during the execution.

In theoretical computer science, a concurrent object has a *name* and a *type*. A *type* is defined by (1) a set of possible values for objects of that type, (2) a finite set of operations through which the objects of that type could be manipulated, (3) a sequential specification describing, for each operation, the condition under which that operation can be invoked, and the effect produced after the operation was executed.

Our dynamic task allocation type DTA supports two operations, i.e, $DoTask(\ )$ and $InsertTask(\ell)$, where ℓ is a task identifier which is unique for each task. The input of the system is a string of $DoTask(\ )$ and $InsertTask(\ell)$ operations to be executed. When a process completes its current operation, it is assigned the next operation in the string.

We formalize the notion of type DTA by specifying the above two operations. For clarity, we define the concepts that a task is inserted and performed. We say a task ℓ is inserted if it is associated to a shared memory location M. we assume that there exists a atomic operation $PutTask(M, \ell)$, and a process could associate task ℓ with memory location M by calling $PutTask(M, \ell)$. It returns *success* if the the task l has already been associated with the location, and returns $failure$ if the location was already associated with another task. Similarly, We assume there exists an atomic operation $TryTask(M)$, and the task ℓ associated with memory location M could be performed atomically by a process by calling $TryTask(M)$. Out of several processes calling $TryTask(M)$, only one receives *success* and the index ℓ of that task, while all the others receive $failure$. This assumption guarantees that only one process could actually perform task l, i.e, task l could be performed at most

once during the executing.

A task is *done* if its index has been returned by a process after a $DoTask(\ )$ call. A task is *available* if it has been inserted but not done yet.

Operation $DoTask(\ )$ performs an available task associated to memory location M by calling $TryTask(M)$. $DoTask(\ )$ performs a task and returns its index l if and only its $TryTask(M)$ operation returns *success*. If there is no available task, then operation $DoTask(\ )$ returns $\perp$.

The $InsertTask(\ell)$ operation inserts task ℓ by calling $InsertTask(M, \ell)$, where M is the memory location task ℓ will be associated to. The $InsertTask(\ell)$ operation inserts a task successfully and returns *success* if and only if its $InsertTask(M, \ell)$ returns *success*.

The type DTA is required to satisfy: $Validity$: If a $DoTask(\ )$ returns ℓ, then before that, an $InsertTask(\ell)$ was executed and returned success. $Uniqueness$: For each task ℓ, at most one $DoTask(\ )$ returns ℓ, i.e, each task is performed at most once.

In addition, the property that every inserted task is eventually done is also a desired progress property of the implementation of type DTA.

Chapter 3

The Adaptive To-Do Tree

Some description here.

Method 1 DoTask()

```
1:  while (true) do
2:    v ← root
3:    if (v.surplus() ≤ 0) then
4:      return ⊥
5:    end if

      /* Descent */
6:    while v is not a leaf do
7:      (x_L, y_L) ← v.left.read()
8:      (x_R, y_R) ← v.right.read()
9:      s_L ← min(x_L − y_L, 2^height(v))
10:     s_R ← min(x_R − y_R, 2^height(v))
11:     r ← random(0, 1)
12:     if ((s_L + s_R) = 0) then
13:       Mark-up(v)
14:     else if (r < s_L/(s_L + s_R)) then
15:       v ← v.left
16:     else
17:       v ← v.rght
18:     end if
19:   end while

      /* v is a leaf */
20:   (x, y) ← v.read()
21:   (flag, l) ← v.TryTask(task[y + 1])
22:   v.CAS((x, y), (x, y + 1)) // Update Insertion Count
23:   v ← v.parent
24:   Mark-up(v)
25:   if flag = success then
26:     return ℓ
27:   end if
28: end while
```

Method 2 Insert(task ℓ)

```
1: while (true) do
2:    v ← root
      /* Descent */
3:    while (v is not a leaf) do
4:       (x_L, y_L) ← v.left.read()
5:       (x_R, y_R) ← v.right.read()
6:       s_L ← min(x_L − y_L, 2^height(v))
7:       s_R ← min(x_R − y_R, 2^height(v))
8:       r ← random(0, 1)
9:       if ((s_L + s_R) = 0) then
10:         Mark-up(v)
11:      else if (r < s_L/(s_L + s_R)) then
12:         v ← v.left
13:      else
14:         v ← v.rght
15:      end if
16:   end while
      /* v is a leaf */
17:   (x, y) ← v.read()
18:   flag ← v.PutTask(task[x + 1], ℓ)
19:   v.CAS((x, y), (x + 1, y)) // Update Insertion Count
20:   v ← v.parent
21:   Mark-up(v)
22:   if (flag = success) then
23:      return success
24:   end if
25: end while
```

Method 3 Mark-up(v)

```
1: if (v is not Null) then
2:    for (i = 0; i < 2; i++) do
3:       (x, y) ← v.read()
4:       (x_L, y_L) ← v.left.read()
5:       (x_R, y_R) ← v.right.read()
6:       v.CAS((x, y), (max(x, x_L + x_R), max(y, y_L + y_R))
7:    end for
8: end if
```

Chapter 4

Analysis

4.1 Correctness

The standard correctness condition for shared memory algorithms is linearizability, which was introduced by Herlihy and Wing in 1990. The intuition of linearizability is that real-time behavior of method calls must be preserved, i.e, if one method call precedes another, then the earlier call must have taken effect before the later one. By contrast, if two method calls overlap, we are free to order them in any convenient way since the order is ambiguous. Informally, a concurrent object is linearizable if each method call appears to take effect instantaneously at some moment between its invocation and response.

In the next section, we will first build the formal definition and terminology for linearizability, and then prove that our concurrent dynamic task allocation object is linearizable.

4.1.1 Definitions and Terminology

Definition 1. ***History****. A history (or execution) H, obtained by processes executing shared memory operations on concurrent objects, is a finite sequence of method invocation and response events.*

A history H may be related to several concurrent objects. When we consider the concurrency behavior of a specific object, we should focus on the subhistory of that object which is defined as follows:

Definition 2. *$H|obj$ of history H is the subsequence of all invocation and response events in H whose object names are all obj.*

If all invocation and response events in a history H have the same object name obj, then

the $H|obj = H$. Thus, in the following discussion, when we discuss the concurrency behavior of a specific objet obj, the history H and $H|obj$ are the same.

We use $inv_H(op)$ to denote the position of the invocation of operation op in history H. Similarly we use $rep_H(op)$ to denote the position of the corresponding response of op in H. Actually, for each operation op, it is not necessary to have a matching response. In this case, we call the operation op is pending and denote it as $rsp_H(op) = \infty$. Otherwise, we call the operation is complete in H.

Definition 3. ***Complete/Imcomplete History****. A history is complete if all its operations are complete. A completion of an incomplete history H is an extension H' of H, such that H' contains exactly the same operations as H but the responses are added for the pending operations in H.*

Let H be a complete history. With each operation op in H we could associate a time interval $I_H(op) = [inv(op), rsp(op)]$. Similarly, for an uncomplete history, we denote the interval with respect to the pending operation op by $I_H(op) = [inv(op), \infty]$.

Definition 4. ***Sequential History****. A history is sequential if the first event in the history is an invocation, and each invocation, except possibly the last one, is immediately followed by a corresponding response.*

It is obvious that a sequential history defines a total order over all operations.

Definition 5. ***Sequential Specification****. The sequential specification S_{obj} for a concurrent object obj is a set of sequential histories on obj.*

We say a sequential history S satisfies the sequential specification S_{obj} of object obj, it means $S \in S_{obj}$. A sequential history S of object obj is valid (sometimes also called legal) if it satisfies the sequential specification of obj.

With the above definitions, we could now define linearization as follows:

Definition 6. ***Linearization****. A history H linearizes to a sequential history S, if and only if S satisfies the following conditions: (1) S and the completion of H have the same operations, (2) sequential history S is valid, and (3) there is a mapping from each time interval $I_H(op)$ to a time point $t_H(op) \in I_H(op)$, such that the sequential history S could be obtained by sorting the operations in H by their $t_H(op)$ values.*

A history is linearizable if and only if there exists a sequential history S that linearizes H. In this case, S is called linearization of H.

We choose one linearization of H and this linearization defines $t_H(op)$. For each operation op in history H, we call the point $t_H(op)$ linearization point of op.

Definition 7. ***Linearizable Object****. A concurrent object is linearizable if every history H of that object is linearizable.*

In the following subsection, we will prove that the concurrent dynamic task allocation object shown in Figure 1 is linearizable.

4.1.2 Analysis and Proofs

By the definitions in Subsection 3.1.1, one way to show a concurrent object obj is linearizable is to prove every history H of obj is linearizable. Thus, we need to identify for each DoTask and InsertTask operation op (i.e, interval $I_H(op)$) in H a linearization point $t_H(op)$, and prove that the sequential history S obtained by sorting these operations according to their $t_H(op)$ satisfies the sequential specification S_{obj} of obj.

We notice that each complete DoTask or InsertTask operation can be associated with a unique task array slot based on the task it removed or inserted. Additionally, the removal and insertion count are both monotonically increasing. Thus, we could associate the node counts with operations which have been propagated to that node.

Now we define "an operation is counted at a node" recursively to formalize the operation propagation.

A DoTask operation is counted at leaf v when the removal count of v is updated with the index of the task array slot where the performed task is located. Symmetrically, an InsertTask operation is counted at v when the insertion count of v is updated with the index of the task array slot where the inserted task is located.

Now we only define DoTask operation is counted at an inner node v because counting an InsertTask operation is symmetric as well.

Recall that the removal count of v is updated though CAS operation (line 6, method 3). Actually there could be more than one operations updating the count with the same value. We linearize all such CAS operations, which update the removal count of v with the same value y. We say for all these operations, only the first one in the linearization order counts the corresponding DoTask operation. In another word, a DoTask operation is counted at an inner node v as soon as the CAS updating operation that counts the DoTask is linearized. Based this definition, no operation will be counted twice at a node.

Please note that, the CAS operation counting the DoTask at node v is not necessary performed by the DoTask operation itself, i.e, suppose process p executes a DoTask operation and has successfully performed task ℓ at certain leaf. Then the CAS operation counting this $p.DoTask()$ at node v could be a different process q as long as q updates the removal count first in the linearization order.

Given the above concepts and properties, we could prove the following result:

Lemma 1. *Let v be a tree node,*

(1) If (x, y) is the return value of $v.read()$, then there exists a set of x InsertTask operations and a set of y DoTask operation that have been counted at node v by the end of the execution of $v.read()$.

(2) If there are x DoTask operations that have been counted at v before the execution of $v.read()$, then the removal count value returned by $v.read()$ is not less than x.

Proof. TBD □

Lemma 2. *Consider a history H and an arbitrary operation op in H, let $t_H(op)$ be the point when op is counted at the root, then $t_H(op)$ is between $inv_H(op)$ and $rsp_H(op)$.*

Proof. Without loss of generality, suppose process p executes DoTask operation op which has performed task ℓ successfully at the leaf. When it reaches the $root$ and executes Mark-up($root$), there will be two cases:

Case 1: It increments the removal count of $root$ successfully via CAS (line 6, method 3) at point τ. If it is the first one in the linearization order of all CAS operation updating of the removal count with the same value, then $t_H(op) = \tau$, therefore $inv_H(op) < t_H(op) < rsp_H(op)$. Otherwise, we could let $t_H(op) = \tau'$, where τ' is the time when the first CAS operation in the linearization order updated the removal count. Thus $t_H(op) < \tau < rsp_H(op)$, Because only if the task has been performed then the removal count of root could be updated. so $t_H(op) > inv_H(op)$. Therefore, in this case, $inv_H(op) < t_H(op) < rsp_H(op)$ holds.

Case 2: It fails to increment the removal count. The CAS operation of p fails if and only if the value of the counts were updated by another process at a point before τ, suppose it is τ'. Please note that, we say the counts were updated, it means the removal count or the insertion count was updated because the two counts are stored in one memory location. Thus, there are two subcases.

Subcase 2.1: If it is the removal count that was incremented at τ', it means the DoTask operation has already been counted by another process. Thus, $t_H(op) \leq \tau' < rsp_H(op)$.

Subcase 2.2: If it is not the removal count but the insertion count that was updated at τ'. We notice that the CAS operation (line 6, method 3) will be repeated by p, during the second iteration, if the CAS of p succeed at τ'', then we could deduce that $t_H(op) \leq \tau''$, therefore $t_H(op) < rsp_H(op)$. If it fails again at point τ'' , then there must be another process updated the counts of root again. This time, the counts of the children will be noticed and the DoTask operation will propagate to the root. We could deduce $t_H(op) < \tau''$. Thus, $t_H(op) < rsp_H(op)$ as well. □

Under the above definitions and properties, we claim the point $t_H(op)$ when op is counted at the root is the linearization point of operation op.

Lemma 3. *The dynamic task allocation object in the above figure is linearizable.*

Proof. Consider an arbitrary history H containing DoTask and InsertTask operations. We should prove for any execution of our algorithm, the total order given by the linearization point is verifies the uniqueness and validity. If H is not complete, then we let all processes that have not finished their operations continue to take steps in an arbitrary order until all operations are completed. Every operation will finally be done is ensured by our computation model and the randomness of our algorithm. This way we obtain a completion H' of H and it suffices to prove H' is linearizable. Thus, to prove this lemma, we should prove the total order obtained by sorting the operations by their $t_H(op)$ values is valid.

The uniqueness is obvious. When multiple processes are calling TryTask(ℓ) at the memory location, only one of them will receive success and the index ℓ of the task, all the other competitor processes will get failure. Task ℓ is performed successfully as long as the value of corresponding memory location is turned to 1. The following process will never repeatedly turn it be to 0 and turn 0 to 1 which is guaranteed by the our semantics of task insertion and removing.

Now we prove the validity, i.e. each task that is performed successfully must have been inserted before. We should prove the insertion operation of a task is always counted at the root before the removal operation. To prove this result holds for the root, we now prove it by induction from the leaf.

At the leaf, this holds because if and only if the insertion count of newly inserted task ℓ has been incremented (line 19, method 2) then the following removal operation could read that (line 20, method 1) to know the available task ℓ at the leaf and then try to perform it. In another word, suppose the task ℓ is inserted but the insertion count is not incremented (i.e. insertion has not been counted yet), then the following removal operation has no way

to know the available task ℓ, perform it and increment the removal count. Thus, the removal will not be counted.

For an arbitrary inner node v, we suppose, by the induction step and lemma 1, the result holds for children $v.left$ and $v.right$. We could notice that any process updates the insertion count and removal count as an atomic operation (line 6, method 3). If some remove operation has been counted at node v, then the corresponding insert operation for that task must have been counted at v simultaneously by the double compare-and-swap operation. Apply this to the root, then validity condition holds. □

4.2 Performance

4.2.1 DoTask Analysis

4.2.2 InsertTask Analysis

4.3 Competitive Analysis

Chapter 5

Conclusions

5.1 Contributions

5.2 Future Work

Appendix A

First Appendix