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.

1.1 The Task Allocation Problem

Task allocation problem, also called do-all problem, is already a foundational problem in distributing computing domain. 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 and shared memory version of the task allocation problem. The dynamic task allocation via asynchronous shared memory problem could be described as follows:

p processes must cooperatively perform a set of tasks in the presence of adversity. The tasks are injected by the adversary dynamically over time.

1.2 Related Work

The task allocation via shared memory was firstly discussed by Kanellakis and Shavartsman[1]. In their paper about PRAM algorithm, they gave a robust parallel solution to the task allocation problem. The data structures they used are four full binary trees and tasks are abstracted as registers in an array. Each cell of the array is initialized with 0, and when it is turned to be 1 implies the abstracted task is performed successfully.

Up to now, there have been many research results focus on the topic. ..

Dan Alistarh and Michael, etc.

However, the above papers about the task allocation problem all focus on the static version, or called one-shot version, where the tasks are available at the beginning, and the execution is done when all these tasks are performed successfully. Dan Alistarh and Michael, etc. considered the dynamic version of task allocation problem, i.e. given p asynchronous processes that cooperate to perform tasks while the tasks are not available but inserted dynamically by the adversary during the execution. In that paper, they gave the first asynchronous shared memory algorithm for the dynamic task allocation, and proved that their algorithm is optimal within logarithmic factors. The main idea is to use a full binary full tree, called To-do tree which is inherited from their last paper, to guide the processes to insert the tasks at random empty memory locations and to pick newly inserted tasks to perform at random.

1.3   Statement of Results

In this thesis, we will introduce a randomized adaptive To-do tree algorithm which is similar to that presented by Dan Alistarh and Michael, etc. in their paper []. The performance of our algorithm depends on the input sequence of tasks but not a fixed value. It has no constraint on the number of tasks presented in the to-do tree. Additionally, our algorithm implemented by the compare-and-swap registers directly which makes the complexity of our algorithm to be quadratic.

In chapter 2, we will formally define the dynamic task allocation problem again and describe out adaptive To-do tree algorithm.

In chapter 3, we will give a framework for the performance analysis of our algorithm.

In chapter 4, we summarize our results, relate them to past work and discuss the open questions that arise from our work.

Chapter 2

The Model and Problem Statement

In this chapter, we will formally describe the system model we are working in and introduce the adaptive To-do tree algorithm, a randomized algorithm that could help us solve the dynamic asynchronous task allocation problem better.

2.1 Standard Model

The computation model we work in is the standard asynchronous shared memory mode with p processes 1, 2 p, where up to p-1 processes may fail by crashing. The processes communicate with each other via the shared registered, which they could perform read, write and compare-and-swap (CAS) operations. Each process has a unique identifier i which is from an unbounded namespace and has a local random number generator to perform coin flip operations.

The mode we discuss is under the control of a strong adaptive adversary. At any point of time, it can see the entire history (all coin flip operations and the returned values). Depending on such history, it decides which processes take the next step. i.e. At any point of time, the adversary knows exactly which step each process will be executing next.

2.2 Problem Statement

( ) Dynamic asynchronous task allocation is the problem in which the p processes cooperatively execute tasks and the task are inserted into the data structure dynamically during the executing.

[What is task] For simplicity, the tasks are still abstracted and associated with shared registers. If a process changes the value of certain register from 0 to 1 successfully, it

means the task associated with this register is performed; if a process changes the value of certain register from 1 to 0, it means the task is inserted successfully and associated with that register. Additionally, we assume each task has a unique identifier l¿=0. One simple implementation of such task identifier uniqueness is to assign each task with an identifier in the form (id, count), where id the id of the process to which the task is assigned and count is the value of a local per-process counter, which is incremented on each newly inserted task.

[Complexity Metric] The complexity measure we use is the total number of operations (i.e. read, write, and CAS) that processes take during an execution.

2.3   The Dynamic Adaptive To-Do Tree

[some concepts and definition, e.g. tree, counts, surplus, space] [Double Compare and Swap] [Intro and detailed explanation of DoTask and InsertTask]

Method 1 DoTask()

1: **while** (true) **do**
2:    $v \leftarrow root$
3:    **if** $(v.surplus() \leq 0)$ **then**
4:       **return** $\perp$
5:    **end if**

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

   /* v is a leaf */
20:   $(x, y) \leftarrow v.read()$
21:   $(flag, l) \leftarrow v.\text{TryTask(task}[y + 1])$
22:   $v.\text{CAS}((x, y), (x, y + 1))$ // Update Insertion Count
23:   $v \leftarrow v.parent$
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:     $v \leftarrow root$

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

    /* v is a leaf */
17:    $(x, y) \leftarrow v.read()$
18:    $flag \leftarrow v.\text{PutTask}(task[x + 1], l)$
19:    $v.\text{CAS}((x, y), (x + 1, y))$ // Update Insertion Count
20:    $v \leftarrow v.parent$
21:    Mark-up(v)
22:    **if** $(flag = \text{success})$ **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:         $(x, y) \leftarrow v.read()$
4:         $(x_L, y_L) \leftarrow v.left.read()$
5:         $(x_R, y_R) \leftarrow v.right.read()$
6:         $v.\text{CAS}((x, y), (max(x, x_L + x_R), max(y, y_L + y_R)))$
7:     **end for**
8: **end if**

Chapter 3

Analysis

3.1 Linearizability

The standard correctness condition for shared memory algorithm is Linearizability [M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3):463-492, 1990.] which was firstly raised by M. Herlihy and J. M. Wing in 1990. An implementation is linearizable if for every execution, there is a total order of all completed operations and a subset of the uncompleted operations in the execution that satisfies the sequential specifications of the object and is consistent with the real-time order of these operations. The basic rule behind Linearizability is that every concurrent history is equivalent to some sequential history. If one method call proceeds another, then the other one mush have taken effect before the later one call. By contrast, if two method calls overlap, then we are free to order them in any convenient way.

The usual way to show a concurrent object implementation is linearizable is to identify for each method a linearization point where the method take effect.

In the following subsection, we will obtain the Linearizability of our dynamic task allocation object by showing that, for each method, the linearization point is the time when this method is counted at the root of the adaptive To-do tree. With the help of the linearization point, we could thus find a total order which verifies (1) validity, each task that is performed successfully must have been inserted before, (2) uniqueness, each task is inserted and performed exactly once.

Now we define a DoTask or InsertTask is counted at a node by noticing that each successful operation can be associated with a unique index of the task array at the leaf corresponding

to the task it removed or inserted.

An operation (DoTask or InsertTask) is counted at a leaf at the point when some process updates the count corresponding to the operation type with the index of the task array slot the operation deals with.

Now we define an operation is counted at an arbitrary inner node v. We could linearize all operations that update the removal count (or insertion count) of v with the same value y (or x). Only the first updating operation in in the sequential order counts the corresponding DoTask (or InsertTask) operation. Namely, a DoTask operation is counted at an inner node v at the point as soon as the updating operation that counts the DoTask is linearized. Symmetrically, a InsertTask operation is counted at an inner node v at the point as soon as the updating operation that counts the InsertTask is linearized. Please note that, the operation updating the counts of v is not necessary performed by the operation that performs that task at the leaf. i.e, suppose process p inserts task l into the task array at certain leaf, but the operation that counts such InsertTask at the node v could be a different process q.

Lemma 1. *Each complete DoTask or InsertTask operation op will finally be counted at each node.*

Sketch. This is ensured by the repetition of double compare and swap operation during the execution of mark-up(v). □

Lemma 2. *The point $t_H(op)$ when op is counted at the root must be between $inv_H(op)$ and $rsp_H(op)$.*

Proof. The fact $t_H(op) > inv_H(op)$ is needless to prove. Without loss of generality, suppose process p executes op and op is a DoTask operation which has performed some task successfully at the leaf. When it walks back to root and executes Mark-up(root), there will be two cases:

Case 1: It increments the removal count successfully via CAS (line 6, method 3) at point τ. If it is the first one in the linearization order of all operations updating the removal count

with the same value, then $t_H(op) = \tau$, therefore $t_H(op) < rsp_H(op)$. Otherwise, we could let $t_H(op) = \tau'$, where τ' is the first one in the linearization order updated the removal count of root. Obviously, $\tau' < \tau$, therefore $t_H(op) < rsp_H(op)$.

Case 2: It fails to increment the removal count. The compare and swap operation of p fails if and only if the value of the insertion and/or removal counts are/is updated at a point before τ, suppose it is τ'. Here are two subcases.

Subcase 2.1: If the removal count is 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 is 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.

9

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. □

3.2 Performance

3.2.1 DoTask Analysis

3.2.2 InsertTask Analysis

3.3 Competitive Analysis

Chapter 4

Conclusions

4.1   Contributions

4.2   Future Work

Appendix A

First Appendix