UNIVERSITY OF CALGARY

Dynamic Task Alloction in Asynchronous Shared Memory

by

Longsheng Zhou

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF Master of Science

DEPARTMENT OF Computer Science

CALGARY, ALBERTA

monthname, 2016

Abstract

TBD

Acknowledgements

TBD

Table of Contents

Chapter 1

Introduction

Asynchronous *task allocation* problem, also called *do-all* problem[5], is defined informally as the problem of n processes in the network, cooperatively performing m independent tasks, in the presence of adversity.

Such cooperation problems consisting of large numbers of tasks by multiple processes is highly related to a broad range of distributed computing problems, such as mutual exclusion [4], consensus problem [11] distributed clocks [3], and shared-memory collect [1].

In shared memory models, the task allocation problem is known as *Write-All* problem, introduced and studied by Kanellakis and Shvartsman [10] and defined as follows: Given a zero-valued array of m elements and n processors, write value 1 into each array location in the presence of adversity.

Following the initial work [10], the task allocation problem was studied in a variety of shared memory settings e.g., [5, 7, 14, 51, 65, 68, 69, 82, 87, 88, 89].

1.1 Related Work

1.2 Statement of Results

Chapter 2

Model of Computation and Definitions

2.1 Asynchronous Shared Memory Model

In this chapter, we will describe our model of computation and give the definitions, which are based on Herlihy and Wing's [8] and Golab, Higham and Woelfel's [7].

The computational model we consider is the standard asynchronous shared memory model with a set $\mathcal{P}$ of n processes, denoted as $\mathcal{P} = [p] = \{0, 1, 2, ..., n-1\}$, where up to $n-1$ processes may fail by crashing. A process may crash at any moment during the computation and once crashed it does not restart, and does not perform any further actions.

Type and Object. A *type* τ is defined as an automaton as follows [6],

$$\tau = (\mathcal{S}, s_{init}, \mathcal{O}, \mathcal{R}, \delta)$$

where $\mathcal{S}$ is a set of states, $s_{init} \in \mathcal{S}$ is the initial state, $\mathcal{O}$ is a set of operations, $\mathcal{R}$ is the set of responses, and $\delta : \mathcal{S} \times \mathcal{O} \to \mathcal{S} \times \mathcal{R}$ is a state transition mapping.

An *object* is an implementation of a type. For each type τ, the transition mapping δ captures the behaviour of objects of type τ, in the absence of concurreny, as follows: if a process applies an operation *opt* to an object of type τ which is in state s, the object may return to the process a response *rsp* and change its states to s' if and only if $(s', rsp) \in \delta(s, opt)$.

History. A *history* H, obtained by processes executing operations on objects, is a sequence of invocation and response events.

An invocation event is a 5-tuple,

$$\text{INV} = (invocation, p, obj, opt, t)$$

where *invocation* is the event type, p is the process executing the operation, obj is the object on which the operation is executed, opt is the operation and t is the *time* when INV happens which is defined as the position of event INV in history H. We also say the event INV is the invocation event of operation opt.

A response event is also a 5-tuple,

$$\text{RSP} = (response, p, obj, rsp, t)$$

where *response* is the event type, p is the process receiving response rsp from an operation on object obj and t is the time when RSP happens which is defined as the position of event RSP in history H.

In the following discussion, we suppose in a history H, the situation that an invocation event $(invocation, p_i, obj_p, opt_0, t_0)$ is followed immediately by another invocation event $(invocation, p_j, obj_q, opt_1, t_1)$ where $i = j$ and $p = q$ will not happen.

We say response event $(response, p_j, obj_q, rsp, t_1)$ *matches* invocation event $(invocation, p_i, obj_p, opt, t_0)$ in history H, if the two events are applied by the same process to the same object, i.e, $i = j$ and $p = q$. In this case, the response event is also called the *matching* response of the invocation event.

An *operation execution* in H is a pair $oe =$ (INV, RSP) consisting of an invocation event INV and its matching response event RSP, or just an invocation event INV with no matching response event, denoted as $oe =$ (INV, *null*).

In the latter case, we say the operation execution is *pending*. In the former case, we say the operation execution is *complete*.

A history H is *complete* if all operation executions in H are *complete*. Otherwise, it is *incomplete*.

If events INV and RSP are applied by process p, then we say operation execution oe = (INV, RSP) is *performed* by process p. Thus, two operation executions performed by the same process on the same project will not interleave in a history H.

We say that an operation *opt* is *atomic* in history H, if *opt*'s invocation event is either the last event in H, or else is followed immediately in H by a matching response event.

If history H is a prefix of history H', then we say history H' is an extension of H. History H' is a *completion* of history H if H' contains all events in H and H' is an extension of H, and each operation execution in H' is complete.

$H|obj$ of history H is the subsequence of all invocation and response events in H on object obj. If all invocation and response events in a history H have the same object name obj, then the $H|obj = H$.

Let H be a complete history. We associate a time interval $I_{oe} = [t_0, t_1]$ with each operation execution oe = (INV, RSP) in H, where t_0 and t_1 are the points in time when INV and RSP happen. Similarly, for an incomplete history, we denote the time interval I_{oe} with respect to a pending operation execution oe = (INV, *null*) by $I_{oe} = [t_0, \infty]$.

Operation execution oe_0 *precedes* operation execution oe_1 in H if the response event of oe_0 happens before the invocation event of oe_1 in H. We say that oe_0 and oe_1 are *concurrent* in H if neither precedes the other.

A history is *sequential* if its first event is an invocation event, and each invocation event, except possibly the last one, is immediately followed by a matching response event.

A *sequential specification* of an object is the set of all possible sequential histories for that object.

A sequential history S is *valid*, if for each object obj, $S|obj$ is in the sequential specification of obj.

Linearization. A history H *linearizes* to a sequential history S, if and only if S satisfies the following conditions:

- S and any completion of H have the same operation executions,
- sequential history S is valid, and
- there is a mapping from each time interval I_{oe} to a time point $t_{oe} \in I_{oe}$, such that the sequential history S is obtained by sorting the operations in H based on their t_{oe} values.

A history is *linearizable* if and only if H linearizes to some sequential history S. In this case, S is called the *linearization* of H. For each operation *opt* in history H, we call time point t_{oe}, which is defined as above, the *linearization point* of *opt*. An object *obj* is linearizable if every history H on *obj* is linearizable.

2.2 Base Objects

In this section, we describe the two base objects, i.e, *read-write register* and *compare-and-swap* (CAS) objects, which will be used in our following discussion. Most implementations of more sophisticated objects use them as the base objects in their implementations and most modern architectures support either read-write registers and CAS objects [9] [12].

Read-Write Register. An object that supports only `read()` and `write(x)` operations is called a read-write register (or just *register*).

Operation `read()` returns the current state of register and leaves the state unchanged. Operation `write(x)` changes the state of the register to x and returns nothing. If the set of states that can be stored in the register is unbounded then we say the register is *unbounded register*; otherwise the register is *bounded register*.

CAS Object. An object that supports `read()` and `CAS(x,y)` operations is called compare-and-swap (CAS) object.

Operation `read()` returns the current state of CAS object, and leaves the state unchanged, while operation `CAS(x,y)` changes the state of the object if and only if the current state is equal to x and then operation `CAS(x,y)` succeeds, and the state is changed to y and *true* is returned. Otherwise, operation `CAS(x,y)` fails, the current state remains unchanged and $false$ is returned.

2.3 Adversary Models for Randomized Algorithms

Randomness. A randomized algorithm is an algorithm where processes are allowed to make random decisions for future steps by calling a special operation called *coin-flip operation*. We also say a process *flips a coin* when it calls this operation.

When a process flips a coin, it receives a random value c from some arbitrary countable set Ω, which is the *coin-flip domain*. The process can then use this random value c in its program for future decisions.

A vector $\overrightarrow{\mathrm{c}} = (c_0, c_1, c_2, ...) \in \Omega^\infty$ is called a *coin-flip vector*. A history H is said to *observe* the coin-flip vector $\overrightarrow{\mathrm{c}}$ if for any integer $i \in \{0, 1, 2, ...\}$, the i-th coin-flip operation in H returns value $c_i \in \Omega$.

For a history H that contains k coin-flip operations, we use $H[k]$ to denote the prefix of H that ends with the k-th invocation of a coin-flip operation. If fewer than k coin-flips occur during H, then $H[k]$ denotes H.

Schedule. In the standard shared memory model, each process executes its program by applying shared memory operations (`read()`, `write(x)`, `CAS(x,y)`, etc) on objects, as determined by their program. Operation executions of concurrent processes can be interleaved arbitrarily.

A *schedule* with *length* k is represented by a sequence of process IDs

$$p = (p_0, p_1, p_2, ..., p_{k-1})$$

where $k \in \{1, 2, 3, ...\}$ and for each $i \in \{0, 1, ..., k-1\}$, $p_i \in \mathcal{P}$.

Consider a schedule $p = (p_0, p_1, p_2, ..., p_{k-1})$. A history H is said to *observe* schedule p if the number of events in H is k, and for each integer $i \in \{0, 1, ..., k-1\}$, the i-th event is applied by process p_i.

Adversary. In a randomized algorithm, the random choices processes make can influence the schedule. To model the worst possible way that the system can be influenced by the random choices, schedules are assumed to be generated by an adversarial scheduler, called the *adversary*.

Mathematically, an adversary is defined as a mapping [7]:

$$\mathcal{A} : \Omega^\infty \rightarrow \mathcal{P}^\infty$$

Given an algorithm $\mathcal{M}$, an adversary $\mathcal{A}$, and a coin-flip vector $\overrightarrow{c} \in \Omega^\infty$, a unique history $H_{\mathcal{M},\mathcal{A},\overrightarrow{c}}$ is generated, such that all processes apply events as dictated by algorithm $\mathcal{M}$, and history $H_{\mathcal{M},\mathcal{A},\overrightarrow{c}}$ observes the schedule $\mathcal{A}(\overrightarrow{c})$ and the coin flip vector $\overrightarrow{c}$.

There are several adversary models with different strengths [2]. In our thesis, we only conside the *adaptive adversary*.

Informally, the adaptive adversary makes scheduling decisions as follows: At any point, it can see the entire history up to that point. This includes all coin-flip operations and their return values up to that point. Depending on this, the adversary decides which process takes the next step.

Adversary $\mathcal{A}$ is *adaptive for algorithm* $\mathcal{M}$ [7] if, for any two coin-flip vectors $\overrightarrow{c} \in \Omega^\infty$ and $\overrightarrow{d} \in \Omega^\infty$ that have a common prefix of length k (i.e, the first k elements of $\overrightarrow{c}$ and $\overrightarrow{d}$ are the same), then we have

$$H_{\mathcal{M},\mathcal{A},\overrightarrow{c}}[k+1] = H_{\mathcal{M},\mathcal{A},\overrightarrow{d}}[k+1]$$

In this case, we say adversary $\mathcal{A}$ is an *adaptive adversary.*

From the above definition, we can see an adaptive adversary cannot use future coin flips to make current scheduling decisions.

2.4 The Dynamic Task Alloction Problem

2.4.1 Task

A *task* is a computation which is assumed to be performed by a single process in constant time[5]. In this thesis, we consider a finite or infinite set of tasks, denoted as $\mathcal{L} = [m] = \{0, 1, ..., m-1\}$, where $m \in \{0, 1, 2, ...\}$.

We assume that each task $\ell \in \mathcal{L}$ to be performed is associated with a *location* M in the data structure. Over time, one location can be associated with multiple tasks.

In this section, we are going to specify the dynamic task alloction problem in terms of a type DTA which suports two types of operations `DoTask` and `InsertTask`, and the properties that an implementation of type DTA must satisfy. But before that, we firstly fix an interface by which processes could perform a task, or insert a new task to data structure.

Operation TryTask(M). A process can perform a task ℓ atomically by calling a special `TryTask(`M`)` operation where M is a location which task ℓ is associated with.

If the location M is associated with a task ℓ, then notification *success* will be returned by

`TryTask(`M`)`. Otherwise, if there is no task associated with location M, then *failure* will be returned.

Operation PutTask(M, ℓ). A process can associate a task ℓ with a location M in the data structure atomically by calling a special `PutTask(`M, ℓ`)` operation.

If location M is not associated with any other task, then `PutTask(`M, ℓ`)` will return *success*. Otherwise, if location M is already associated with another task ℓ', then $failure$ is returned by `PutTask(`M, ℓ`)` call.

2.4.2 The Type DTA

The type DTA supports two types of operations. The `DoTask()` operation performes a task and returns the identifier of that task, while the `InsertTask(`ℓ`)` operation associates task ℓ with a location in the data structure to be performed. Now we describe the sequential specification as follows.

Operation DoTask(). The aim of the operation `DoTask()` is to find a location M which is associated with a task ℓ in the data structure and then perform task ℓ by calling the atomic operation `TryTask(`M`)`.

Every `DoTask()` operation may perform several `TryTask` operations with different locations as the arguments. Once a `TryTask` operation succeeds, `DoTask()` terminates and the task identifier ℓ is returned. Otherwise, `DoTask()` never terminates and keeps calling `TryTask(`M`)` repeatedly. If there is no task in the data structure, then `DoTask()` returns $\perp$.

A task ℓ is said to be *performed* by a process if the process has completed a `DoTask()` call which returns the task identifier ℓ.

Operation InsertTask(ℓ). The goal of the `InsertTask(`ℓ`)` operation is to find a location M in the data structure and associates task ℓ with M by executing operation `PutTask(`M, ℓ`)`.

Every `InsertTask(`ℓ`)` operation may perform several `PutTask` operations with different locations as the arguments. Operation `InsertTask(`ℓ`)` terminates once a `PutTask` operations succeeds and then location M will be returned by `InsertTask(`ℓ`)`. Otherwise, `InsertTask(`ℓ`)` never terminates and keeps calling `PutTask` operations repeatedly.

We say task ℓ is associated with a location M or inserted into the data structure or task ℓ is *available* to perform if a process has completed the `InsertTask(`ℓ`)` call and the location M is returned, but task ℓ has not been performed yet.

With the above sequential specification, we can see an algorithm that access an instance of an object of type DTA must satisfy the following properties: evey performed task must be inserted (*validity*) and each task is performed exactly once (*uniqueness*).

2.4.3 Progress Conditions

Condition 1. *Every inserted task is eventually performed.*

Chapter 3

Data Structure and Implementation

Data Structure

The Implementation of Type DTA

Method 1: `DoTask()`

```
while true do
    v ← root;
    if v.surplus() ≤ 0 then
        return ⊥;
    end
    /* Descent */;
    while v is not a leaf do
        (x_L, y_L) ← v.left.read();
        (x_R, y_R) ← v.right.read();
        s_L ← min(x_L − y_L, 2^{height(v)});
        s_R ← min(x_R − y_R, 2^{height(v)});
        r ← random(0, 1);
        if (s_L + s_R) = 0 then
            Mark-up(v);
        else if r < s_L/(s_L + s_R) then
            v ← v.left;
        else
            v ← v.rght;
        end
    end
    /* v is a leaf */;
    (x, y) ← v.read();
    (flag, l) ← v.TryTask(task[y + 1]);
    /* Update Insertion Count */;
    v.CAS((x, y), (x, y + 1));
    v ← v.parent;
    Mark-up(v);
    if flag = success then
        return ℓ
    end
end
```

Method 2: `InsertTask(ℓ)`

```
while true do
    v ← root;
    /* Descent */;
    while v is not a leaf do
        (x_L, y_L) ← v.left.read();
        (x_R, y_R) ← v.right.read();
        s_L ← 2^{height(v)} − min(x_L − y_L, 2^{height(v)});
        s_R ← 2^{height(v)} − min(x_R − y_R, 2^{height(v)});
        r ← random(0, 1);
        if (s_L + s_R) = 0 then
            Mark-up(v);
        else if r < s_L/(s_L + s_R) then
            v ← v.left;
        else
            v ← v.rght;
        end
    end
    /* v is a leaf */;
    (x, y) ← v.read();
    flag ← v.PutTask(task[x + 1]);
    /* Update Insertion Count */;
    v.CAS((x, y), (x + 1, y));
    v ← v.parent;
    Mark-up(v);
    if flag = success then
        return success
    end
end
```

Method 3: `Mark-up(v)`

```
if v is not null then
    for (i = 0; i < 2; i + +) do
        (x, y) ← v.read();
        (x_L, y_L) ← v.left.read();
        (x_R, y_R) ← v.right.read();
        v.CAS((x, y), (max(x, x_L + x_R), max(y, y_L + y_R));
    end
end
```

Chapter 4

Analysis

4.1 Correctness Proof

By the definitions in Subsection 3.1.1, one way to show an 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: (under work...almost done)

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. $\square$

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. $\square$

4.2 Performance Analysis

DoTask Analysis

InsertTask Analysis

4.3 Competitive Analysis

Chapter 5

Conclusions and Future Work

Bibliography

[1] Miklos Ajtai, James Aspnes, Cynthia Dwork, and Orli Waarts. A theory of competitive analysis for distributed algorithms. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 401–411. IEEE, 1994.

[2] James Aspnes. Randomized protocols for asynchronous consensus. *CoRR*, cs.DS/0209014, 2002.

[3] Yonatan Aumann and Michael O Rabin. Clock construction in fully asynchronous parallel systems and pram simulation. In *Foundations of Computer Science, 1992. Proceedings., 33rd Annual Symposium on*, pages 147–156. IEEE, 1992.

[4] Michael Bender, Seth Gilbert, et al. Mutual exclusion with o (logˆ 2 log n) amortized work. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 728–737. IEEE, 2011.

[5] Chryssis Georgiou. *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity.* Springer Science & Business Media, 2007.

[6] Wojciech Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. Constant-rmr implementations of cas and other synchronization primitives using read and write operations. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 3–12, 2007.

[7] Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 373–382. ACM, 2011.

[8] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[9] Itanium. *Intel Itanium Architecture Software Developers Manual Volume 1: Application Architecture Revision 2.1.* Intel Corporation, October 2002.

[10] Paris C Kanellakis and Alex A Shvartsman. Efficient parallel algorithms can be made robust. *Distributed Computing*, 5(4):201–217, 1992.

[11] Nancy A Lynch. *Distributed algorithms.* Morgan Kaufmann, 1996.

[12] David L Weaver and Tom Gremond. *The SPARC architecture manual.* PTR Prentice Hall Englewood Cliffs, NJ 07632, 1994.