How to Allocate Tasks Asynchronously

Dan Alistarh
EPFL

Michael A. Bender
Stony Brook University and Tokutek, Inc.

Seth Gilbert
NUS

Rachid Guerraoui
EPFL

Abstract

Asynchronous task allocation is a fundamental problem in distributed computing, in which p asynchronous processes must execute a set of m tasks. Also known as *write-all* or *do-all*, this problem has been studied extensively, both on its own and as a key building block for various distributed algorithms.

In this paper, we break new ground on this classic problem: we introduce the To-DoTree concurrent data structure, which improves on the best known randomized and deterministic upper bounds. In the presence of an adaptive adversary, the randomized To-DoTree algorithm has $O(m+p\log p\log^2 m)$ work complexity. We then show that there exists a deterministic variant of the To-DoTree algorithm with work complexity $O(m + p\log^5 m\log^2 \max(m, p))$. For all values of m and p, our algorithms are within log factors of the $\Omega(m + p\log p)$ lower bound for this problem.

The key technical ingredient in our results is a new approach for analyzing concurrent executions against a strong adaptive scheduler. This technique allows us to handle the complex dependencies between the processes' coin flips and their scheduling, and to tightly bound the work needed to perform subsets of the tasks. We believe this technique will be useful in the analysis of other, more complex, concurrent data structures.

1 Introduction

How do we efficiently allocate a set of tasks to a set of processes? This is one of the foundational questions in computer science.

The question is particularly challenging when there is *irregularity*, e.g., when different compute nodes vary in their speed, memory, and level of robustness. As the workload changes, processes may end up with uneven loads, which can throttle performance. For large systems, irregularity is the common case.

Another challenge for task allocation is decentralized scheduling. In some situations, centralized scheduling proves too expensive, either because the system is too large or the granularity of the tasks is too small. In either case, the system needs a decentralized scheduler.

In this paper, we consider *asynchronous task allocation*, a general form of the allocation problem in which processes are asynchronous—and thus behave irregularly—and process scheduling is distributed. A set of p processes cooperates to execute all m tasks, ending up with some "certificate" that all work is completed. Tasks are idempotent, meaning that a task may be executed more than once. Process speeds are governed by an *adaptive adversary*, who knows everything about the current state of the system, but cannot predict the outcome of future coin tosses. We assume the standard *asynchronous shared-memory model*, in which processes communicate by reading and writing to atomic registers. There is no centralized scheduler. Instead, processes coordinate through shared memory to distribute the tasks.

Two metrics are important: the *total work*, that is, the number of steps (reads, writes, and random coin flips) summed over all p processes, and the number of *tasks executed*.

A brief history. Asynchronous task allocation (also called *do-all* [16], *write-all* [18], or *certified write-all* [3]) has been recognized as one of the central problems in distributed computing for decades; the book by Giorgiou and Shvartsman [16] gives a detailed history of this problem. The shared-memory version was introduced over twenty years ago by Kanellakis and Shvartsman [18] in the context of fault-tolerant PRAM computation. In this formulation, tasks are abstracted as shared registers in an array; each register is initially set to 0 and must be flipped to 1. There have been many subsequent papers on the topic, e.g. [3, 13, 15, 22, 24, 25]. Asynchronous task allocation is also the dual of *distributed collect* [2], in which p processors need to aggregate values from m registers. Both task allocation and collect have been used for solving other fundamental distributed problems, such as dynamic load balancing [17], mutual exclusion [10], atomic snapshots [1], consensus [6], renaming [12], distributed phase clocks [8], and PRAM simulation [20].

The last twenty years have seen a long-standing quest to establish the complexity of asynchronous task allocation [3, 13, 15, 22, 24, 25]. Various structures such as low-contention permutations [3] and expander-based progress graphs [15] were developed to this end. This line of work has continually improved the bound on total work, yielding beautiful and technically sophisticated results.

Results. In this paper, we break new ground on this classic problem by taking an alternative approach. We view our algorithm as a randomized protocol from beginning to end. We follow the simple idea that when a process selects a task to execute, it should select randomly and nearly uniformly from the set of remaining tasks. This basic philosophy is natural and appears periodically in the literature [7, 9, 11, 26]. The algorithmic challenge is to build a distributed data structure to perform the selection efficiently. We first analyze the randomized behavior of the resulting data structure. The key technical aspect of our result is showing that the randomness can be isolated to the beginning of the execution: the adversary gets nontrivial extra power from knowing the coin-flip outcomes in advance, but we show that this power is limited. Moving all randomness to the beginning implies, via the probabilistic method, that a deterministic solution exists.

Our main results are as follows:

- We give a randomized algorithm for an adaptive adversary, which we call the To-DoTree algorithm. To-DoTree task allocation uses $O(m + p \log p \log^2 m)$ work, with high probability.
- We show there exists a deterministic To-DoTree algorithm with work $O(m+p \log^5 m \log^2 \max(m, p))$.

setting	deterministic (general m, p)		deterministic ($p = m$)		randomized (strong adversary)	
previous bounds	$O(m + p^{2+\epsilon})$	[22]	$O(m \log^{18} m / \log^3 \log m)$	[15]	$O(m \log m)$	[3]
this paper	$O(m + p \log^5 m \log^2 \max(m, p))$		$O(m \log^7 m)$		$O(m + p \log p \log^2 m)$	
lower bounds	$\Omega(m + p \log p)$	[13]	$\Omega(m \log m)$	[13]	$\Omega(m + p \log p)$	[13]

Table 1: Summary of results and relation to previous work.

The To-Do Tree algorithm improves on existing algorithms in both randomized and deterministic settings; see Table 1. In the worst-case, existing algorithms perform a polynomial factor more work than is optimal. By contrast, the To-Do Tree algorithm runs within a polylogarithmic factor of optimal for all values of m and p. The algorithm is optimized both for $m \geq p$, the common case in most real systems, and $p > m$, a case arising in sublogarithmic shared-memory distributed algorithms; see e.g., [10].

Another advantage of our approach is its relative simplicity. Our deterministic algorithm is precisely the To-Do Tree with the random choices determined in advance. Each process gets its own string of heads and tails that it reads whenever it needs a coin flip. We prove that that we can find good strings, thus demonstrating the existence of a deterministic To-Do Tree algorithm. In fact, if the strings are chosen randomly, the probability that we do not end up with a deterministic algorithm is polynomially small in m and exponentially small in p. Waxing philosophical, this means that for a sufficient choice of constants, the probability that any given random bit string does *not* yield a deterministic solution is vanishingly small compared to the probability that this (or nearly any other paper) contains an unrecoverable bug [27].

To-Do Tree task allocation applies both to the case where tasks are small and scheduling overhead is critical, and to the case where tasks are large. Indeed, even modest-size tasks (e.g., of size $\text{polylog}(m)$) may dominate the scheduling cost; see [7] for a theoretical discussion of such instances.

For larger tasks it makes sense to consider the number of *tasks executed* as a metric—recall that tasks may get executed redundantly. (This metric is related to work sharing with *at-most-once* semantics [21].) The *redundant-task ratio* is the number of tasks executed divided by m. The lower bound of $\Omega(m+p\log p)$ from [13] applies here, meaning that the redundant-task ratio has an $\Omega(1 + p \log p/m)$ lower bound.

The randomized and deterministic variants execute $O(m+p\log p)$ and $O(m+p\log^5 m \log^2 \max(m, p))$ tasks, respectively. Thus, our randomized algorithm has redundant-task ratio $O(1 + p \log p/m)$, which is optimal, and our deterministic algorithm has redundant-work ratio $O(1 + p \log^5 m \log^2 \max(m, p)/m)$.

Intuition behind To-Do Trees. Consider an algorithm in which processes randomly choose tasks to perform. The adaptive adversary can "block" a task j from getting executed by deciding to stop any process that tries to run it. Task j is executed only once all p processes are poised over j and the adversary has no choice but to let some process proceed. As long as there is no data structure to guide the choice of task, and $p = \Omega(\log m)$, this strategy needs $\Omega(mp)$ work to execute all tasks.[1]

Motivated by this example, we use a concurrent data structure, the To-Do Tree, to guide processes towards incomplete tasks. A To-Do Tree is a complete binary tree, where tasks reside in the leaves. The internal nodes of the tree record the number of descendent leaves having unexecuted tasks; see Figure 1.

To select its next task, a process starts at the root and walks down the tree, deciding whether to turn left or right at every internal node based on the number of unexecuted tasks at each of the two children. Specifically, if an internal node records that there are x unexecuted leaves in the left subtree and y in the right subtree, then the walk goes left with probability $x/(x + y)$ and right with probability $y/(x + y)$. Once

[1]This example also illustrates the difference in capabilities between the adaptive and oblivious adversaries. This randomized strategy works fine against an oblivious adversary, because the oblivious adversary can do little to block progress, and after $O(m \log m)$ random choices, all tasks are completed.

the leaf is executed, the process walks back up the tree, updating the counters in the internal nodes.

Structurally, the To-Do Tree is built in similarly to the shared-memory counter from [4], augmented with a random-leaf-search operation. The counters at internal nodes of the To-Do Tree are maintained using the "max-register" construction from [4]. The To-Do Tree can also be seen as an augmented progress tree. The biggest distinction is that we choose leaves based on a biased root-to-leaf walk rather than, say, choosing a random leaf. What we find surprising about the To-Do Tree is that, despite its similarity to other structures, it delivers much lower work bounds and task-executed bounds than have previously appeared in the literature.

The technically most involved result in this paper is the analysis of the deterministic To-Do Tree. In a rough sense, we want to prove that a deterministic algorithm exists by bounding the randomized algorithm's error probability (at most $1/2^{\Theta(p \text{ polylog } m)}$), despite all the random choices being made in advance, and then multiplying by the total number of possible schedules. We then use the probabilistic method to show that a deterministic algorithm must exist.

There are serious obstacles with this approach. The first obstacle is that there are simply too many possible schedules, which overwhelms the error probability. The second more threatening obstacle is that the same set of coin flips can guide processes to different leaves, depending on the adversary's decisions. The adversary therefore has nontrivial power to govern the algorithm dynamics. The dependencies are wrong for optimistically using some kind of balls-and-bins argument to show that in some number of operations, a given number of tasks have been executed. In summary, the challenge is to find a way to make this argument go through even though there are too many schedules and the adversary can use its knowledge of the coin flips to affect the execution of the algorithm.

Previous approaches. Deterministic solutions to the task allocation problem are generally based upon several common ideas. To certify when all tasks have been executed, the processes collectively maintain a so-called *progress tree* [3,13] or some other kind of data structure for tracking work. Each process maintains its own permutation of the m tasks and executes these tasks in the permutation order. One can view this approach as generalizing the simple case where $p = 2$: one process executes the tasks in order $1, 2, 3, \ldots$, while the other executes the task in order $m, m-1, m-2, \ldots$, and at some point in the middle, the two processes meet. When there are $p > 2$ processes, the trick is to choose permutations to cover all the tasks, regardless of how the operations interleave, while simultaneously avoiding too much redundant work. More recently, researchers have used expanders to give algorithms with better work complexity.

The most efficient deterministic task allocation algorithm for general m and p, given by Kowalski and Shvartsman [22], uses $O(m + p^{2+\epsilon})$ total work. It uses a collection of permutations with low contention; to date, it is not known how to construct such permutations in polynomial time, therefore their algorithm is not explicit. The most efficient explicit algorithm was given by Malewicz [24], using $O(m + p^4 \log m)$ work. These last two algorithms achieved optimal $O(m)$ work for $p = O(m^{1/(2+\epsilon)})$ and $p = O((m/\log m)^{1/4})$, respectively. By contrast, our algorithm is work-optimal for a wider range of $p = O(m/\text{polylog } m)$.

For the special case when $p = m$, Chlebus and Kowalski [15] gave a non-explicit deterministic algorithm with complexity $O(m \log^{18} m/(\log \log m)^3)$ and an explicit variant with complexity $m2^{O(\log^3 \log m)}$. This algorithm and its explicit variant are based upon expander constructions. When applied to $m > p$ their approach yields $O((m+p)\text{ polylog}m)$ work. Earlier deterministic algorithms for the task allocation problem appear in, e.g. [3, 17, 18]. In contrast to these upper bounds, there is a $\Omega(m + p \log p)$ lower bound [13], which holds for deterministic algorithms and for randomized algorithms with a strong adversary.

Randomized solutions to the task allocation problem are based upon the idea of having processors select tasks randomly from the entire set of tasks or from the subset of tasks that remain to be executed. Most previous randomized algorithms for the task allocation problem assumed that the adversary is *oblivious*; an oblivious (weak) adversary knows the system state when the algorithm begins, but in contrast to the adaptive adversary, cannot see the outcomes of coin tosses.

Anderson and Woll [3] gave a randomized algorithm that works against an adaptive adversary. The algo-

rithm assumes p processes, $m = p^2$ tasks, and runs in $O(m \log m)$ work. The algorithm is based on random permutations on a special kind of progress tree; derandomized variants have been studied in [14]. Their strategy, as written, does not extend to general m and p. Since the algorithm has complexity $\Theta(m \log m)$ and $m = p^2$, it runs within a log factor of optimal. The randomized algorithm of Martel et al. [25], which runs against an oblivious adversary, performs work $O(m)$ with high probability, using $p = m/\log m$ processes. Thus, it achieves optimal work for this choice of p and a weak adversary. Other randomized algorithms for the oblivious adversary include [8, 19]. For a detailed overview of research on this problem, we refer the reader to the book by Giorgiou and Shvartsman [16].

Outline. Section 2 formalizes the model and definitions used in the paper. Section 3 presents our To-Do Tree algorithm for asynchronous task allocation. Sections 4 and 5 analyze the randomized and derandomized versions of the To-Do Tree algorithm.

2 Definitions and Notations

Model. We assume the classic asynchronous shared-memory model [5, 23] with p processes $1, 2, \ldots, p$, where up to $t < p$ processes may fail by crashing. Each process has a distinct initial identifier i. Processes communicate through atomic read and write operations on registers. The scheduling of the processes' steps and crashes is controlled by a strong adaptive adversary. At all times the adversary knows the state of the entire system, including the results of random coin flips, and can adjust the schedule and the failure pattern accordingly. Algorithms that are correct in this setting are called *wait-free*.

Problem statement. *Asynchronous task allocation* means enabling p processes to execute m distinct tasks, labeled $\{1, \ldots, m\}$. The traditional *work complexity* metric measures the total number of shared-register operations that processes perform during an execution. We also analyze the *tasks executed* metric, counting the total number of times that tasks are executed. Note that all m tasks must be executed at least once, but some tasks may get executed redundantly. The *redundant-task ratio* is tasks executed divided by m.

3 The To-Do Tree Algorithm

In this section, we describe the To-DoTree, a randomized algorithm that solves the asynchronous task allocation problem. In Section 4, we prove its correctness in the presence of an adaptive adversary. In Section 5, we discuss a variant in which all the random choices are made in advance, and show that there exists a deterministic To-Do Tree algorithm.

Min-Registers. A basic building block of our To-Do Trees is a *min-register*, a shared memory object that supports two operations: read and write-min. When the execution begins, each min-register stores a default initial value. During the execution, the value v stored by the min-register only decreases. A write-min(v') operation writes value $v' \geq 0$ only if $v' < v$; otherwise, the min-register remains unchanged. A read operation returns the value currently stored by the min-register. A min-register can be implemented in the same manner as a max-register, described in [4], where every value read and written is simply subtracted from the default initial value of the min-register. (Note that we do not rely on min-registers being linearizable, only that every read operations returns a value no greater than every "preceding" write-min operation.) We present the max/min-register construction in detail in Appendix A.

To-Do Trees. A *To-Do Tree* is a complete binary tree in which each leaf represents a fixed number of tasks. Initially, we analyze a To-Do Tree with m leaves and one task per leaf; for the final result, we use a To-Do Tree with $m/\log m$ leaves and $\log m$ tasks per leaf. (We assume, without loss of generality, that m or $m/\log m$ is a power of 2, as is convenient.)

```
1  treewalk⟨root⟩;
2      v ← root ;
3      if v.MinReg.value = 0 then return ;
       /* Descent */
4      while v is not a leaf do
5          x ← value of min register at left child of v;
6          y ← value of min register at right child of v;
7          if x + y = 0 then goto Mark-up;
8          r ← random(0, 1) ;
9          if r < x/(x + y) then v ← v.left;
10         else v ← v.right;
       /* v is a leaf */
11     execute work at v;
12     v.MinReg.write(0);
13     v ← v.parent;
       /* Mark-up */
14     while v is not null do
15         x ← value of min register at left child of v;
16         y ← value of min register at right child of v;
17         v.MinReg.write(x + y) ;
18         v ← v.parent;
19     return ;
```

Algorithm 1: Pseudocode of the treewalk procedure.

Each node in the To-Do Tree contains a min-register. At the start of the execution, a min-register at node v is initialized with the number of leaves in the sub-tree rooted at v, i.e., with the value $2^{\mathsf{height}(v)}$.

Work on a To-Do Tree. When a process is available to do work, it proceeds to perform a treewalk, described in Algorithm 1. It begins the treewalk by reading the min-register at the root. If the value returned is 0, then all the work is complete, and it returns success.

Otherwise, the treewalk begins a *descent* at the root, in which it repeatedly performs the following steps: (i) it reads value x from the min-register of the left child of the current node; (ii) it reads value y from the min-register of the right child of the current node; (iii) it flips a random coin $r \in [0, 1)$; and (iv) it proceeds to the left child if $r < x/(x + y)$ and to the right child otherwise.

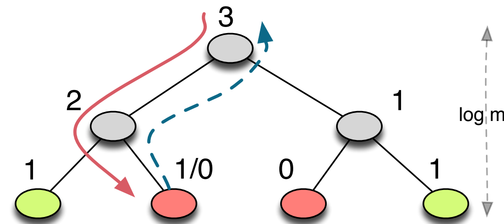

Figure 1: A simple example of a tree-walk operation on a To-Do Tree. The walk descends to reach the red leaf, and then marks up, updating the node min registers. The second red leaf has been counted at the root, as the corresponding min registers have already been updated.

The descent procedure terminates either when the process's treewalk reaches a leaf, or when the process reads both $x = 0$ and $y = 0$ at an internal node. If the descent reaches a leaf, the process performs all the tasks associated with that leaf. After the descent completes and any tasks have been completed, the process begins a *mark-up* procedure from the last node reached during the descent: (i) it moves to the

parent of the current node; (ii) it reads value x from the min-register of the left child of the current node; (iii) it reads value y from the min-register of the right child of the current node; and (iv) it performs a write-min$(x + y)$ operation on the min-register of the current node. The mark-up procedure terminates after the root has been reached and updated.

4 Randomized Analysis

It is easy to see that the To-Do Tree ensures that eventually every task is completed. In this section, we analyze its performance, showing that, on termination, the total number of steps taken and tasks executed is $O(m + p \log p \log^2 m)$; for an alternate variant, the total number of tasks executed is $O(m + p \log p)$. In this extended abstract, we focus on the case where $m \geq p$. The case where $m < p$ follows from an identical analysis.

4.1 Preliminaries

Consider an arbitrary execution. A leaf is *marked* when a treewalk first reaches it, and it is *completed* when it is "accounted for" at the root, and hence we refer to the number of remaining leaves as the value most recently read or writen from the min-register at the root by a completed write-min/read operation.

We divide the execution into two epochs, where the first epoch contains all the steps where there are $\geq 2p$ remaining leaves, and the second epoch contains all the steps where there are $< 2p$ remaining leaves. Each epoch is subdivided into *phases*. The phases are defined such that: in the first epoch, the number of remaining leaves decreases by exactly p in each phase; in the second epoch, the number of remaining leaves decreases by a factor of 2 in each phase.

Claim 1 *There are at most $m/p + \log(2p)$ phases in total.*

We say that treewalk is *complete* in phase k if it begins and ends in phase k. The remainder of this section is dedicated to showing that there are $O(p)$ complete treewalks per phase. (It is immediately clear that there are $\leq p$ treewalks that cross the boundary between every pair of consecutive phases, as there are only p processes active at any given time.)

For a specific write-min and a specific read operation, there is a natural notion of ordering that arises from the implementation[2]. We say that a leaf has been *counted* at an ancestor node v if for some path P from the leaf to v there is a sequence of read and write-min, where each read at the preceding node comes before the write-min at the next node, propagating the count up the tree. It is counted at v as soon as the write-min at v completes, or as soon as any read operation that comes after the last write-min completes.

Lemma 2 (Propagation) *Let z be a tree node at height h of the tree. If a read of the min-register at node z returns value v, then:*

- *There exists a set L_z of $(2^h - v)$ leaves that have been counted at z by the end of the read operation;*
- *If a read of the min-register at z begins after v leaves have been counted at z, then it returns value at most $2^h - v$;*
- *There exists a set F_z of v leaves in the subtree rooted at z such that no leaf in F_z has been counted at z by the beginning of the read operation.*

[2]Specifically, a read operation walks a tree from root-to-leaf, and a write-min operation walks a tree from leaf-to-root. Whichever operation first accesses the lowest node shared by both walks comes first.

Proof. The claim follows by induction on the height of node z in the tree. For a leaf, the claim is trivial: a leaf is, by definition, counted at itself immediately upon its min-register being written, and the value of this register is 0. Assume that the claim holds for all nodes of height $< h$ in the tree, and consider some node z of height h. If a read at z returns value v, then some prior or concurrent write-min operation ψ has written value v. Prior to ψ, as part of the propagation step, there must have been two read operations ϕ_1 and ϕ_2 of the children of z that returned values v_1 and v_2 where $v = v_1 + v_2$. Since the children of z are of height $h - 1$, we can conclude by the inductive hypothesis that there are $2^{h-1} - v_1$ and $2^{h-1} - v_2$ leaves counted at the children of z (respectively) by the end of ϕ_1 and ϕ_2, and hence by the end of the read operation at z; moreover, we can identify the two sets and take their union. This concludes the first claim.

The second claim follows similarly: If v leaves have been counted at z, then there was a previous propagation step which performed a read on the two children of z at some point after $2^{h-1}-v_1$ and $2^{h-1}-v_2$ leaves were counted at the children, respectively, where $v = v_1 + v_2$. (This follows from the definition of a leaf being counted.) Inductively, this means that the read operations at the children return values at least v_1 and v_2, and hence the propagation step writes a value of at least v to z. The second claim then follows from the properties of a min-register. Finally, the third claim is the counterposition of the second. □

This implies that the algorithm terminates if and only if all the work is done.

4.2 Phase Notation

Fix a phase $i \geq 1$. Let B_i be the set of complete treewalks in phase i. Define U_i to be the set of leaves that are not counted at the root at the beginning of the phase. Define S_i to be the set of leaves that are marked by operations in B_i.

Define the reference tree RT_i as a copy of the To-Do Tree where: every leaf that is counted at the root at the beginning of phase i is marked; every other leaf is not marked; and every min-register in the tree reflects exactly the number of unmarked leaves in its subtree.

We define a set W_i as follows: for each treewalk in $t \in B_i$, for each node z that t traverses during the descent: if t reads a value v at z that is smaller than the value v' at the same min-register in the reference tree RT_i, then there must exist a set L of leaves of size $v' - v$ that have been marked during this phase (and counted at z), as per Lemma 2; this set of leaves L is added to W_i.

The key property, which follows from the definition of B_i, S_i, and W_i, is as follows:

Lemma 3 *For each $i \geq 1$, each leaf in $W_i \cup S_i$ is counted at the root by the end of phase i. Also, no leaf may be included in $W_i \cup S_i$ and $W_j \cup S_j$ for distinct phases i and j.*

We now argue that if B_i contains $\Theta(p)$ treewalks, then, with high probability, $S_i \cup W_i$ contains at least $\Theta(p)$ leaves and hence the phase is complete.

4.3 Treewalk Analysis

Fix a set $V \subseteq U_i$ of unmarked leaves. Let e_t be the event that treewalk t chooses a leaf in V. Order the treewalks according to the order in which they took the last step of their descent. Let P_t be the event that some leaf in V is counted by some treewalk in B_i that precedes t. We lower bound the probability of (e_t **or** P_t), given that an arbitrary subset of other leaves have been concurrently marked.

We proceed by induction, calculating the probability after each step that a treewalk arrives at a node in V. We show that for each treewalk in B_i, the probability of choosing a leaf in V is at most $1 - |V|/|U_i|$.

For every node z, define F_z to be the set of leaves in the subtree rooted at z that are not counted at z at the end of the read by the treewalk of the min-register at node z.

Claim 4 *For every node z on treewalk $b \in B_i$, the probability that the treewalk b, starting from node z, counts a leaf in V, or that event P_b occurs, assuming an arbitrary set of concurrently marked/counted leaves, is $\geq |F_z \cap V|/|F_z|$.*

Proof. We proceed by induction on h, the height of node z. If z is a leaf (i.e., $h = 0$), then the claim follows trivially. If the height is $h > 0$, then we consider several cases. First, if all the leaves in F_z are also in V, then the probability of counting a leaf in V is 1. Second, if all the leaves F_z are not in V, then the probability $|F_z \cap V|/|F_z|$ is 0, and again the claim holds.

Consider the case where some leaves in F_z are in V, while others are not. Let x be the right child of z, and y be the left child of z.

If treewalk b gets stuck at node z, i.e. it reads 0 from the min register at x and 0 from the min register at y. Then, by Lemma 2, some leaf in V is counted by b with probability 1.

Assume treewalk b does not get stuck at z. Let v_x be the value read from the min-register at node x, and let v_y be the value read from the min-register at node y. By the induction step, the probability of arriving at a node in V, or P_b holding is:

$$\frac{v_x}{v_x + v_y} \cdot \frac{|F_x \cap V|}{|F_x|} + \frac{v_y}{v_x + v_y} \cdot \frac{|F_y \cap V|}{|F_y|}.$$

On the other hand, by definition, we have that $v_x = |F_x|$, and $v_y = |F_y|$, and hence this is:

$$\frac{|F_x \cap V| + |F_y \cap V|}{|F_x| + |F_y|}.$$

Since the read on the min-register at z finishes *before* the reads at x and y, we get that $v_z \geq v_x + v_y$, i.e., $|F_z| \geq |F_x| + |F_y|$. (Notice that an arbitrary set of leaves may be concurrently marked and/or counted.) If some leaf in V is counted at z or x or y, then some leaf in V is counted with probability 1. Otherwise, since no leaf in V is counted at z, we get that $|F_z \cap V| = |F_x \cap V| + |F_y \cap V|$. Therefore:

$$\frac{|F_x \cap V| + |F_y \cap V|}{|F_x| + |F_y|} \geq \frac{|F_z \cap V|}{|F_z|}.$$

This concludes the proof of this claim. □

This immediately implies the following:

Lemma 5 *For every treewalk $b \in B_i$, the probability that b does not count a leaf in V and that every other preceding treewalk does not count a leaf in V is $\leq 1 - |V|/|U_i|$, assuming an arbitrary set of leaves marked and counted concurrently.*

Proof. Let F be the set of leaves uncounted at the root when the treewalk b reads the min-register at the root at the beginning of its descent. By Claim 4, the probability that b descends to a leaf in V or that some leaf in V is marked is $\geq |F \cap V|/|F|$. If some leaf in V is marked, then the probability is 0 and the claim holds. Otherwise, $|F \cap V|/|F| = |V|/|F| \geq |V|/|U_i|$ (as $F \subseteq U_i$). From this the claim follows. □

Finally, we calculate the probability that no treewalk in B_i marks a leaf in V:

Lemma 6 *For a fixed set of leaves V uncounted prior to phase i, the probability that no treewalk in B_i counts a leaf in V is $\leq (1 - |V|/|U_i|)^{|B_i|}$.*

Proof. First, we observe that the probability that no treewalk in B_i counts a leaf in V is:

$$\Pr(\bar{e}_1 \wedge \bar{e}_2 \wedge \ldots) \leq \Pr((\bar{e}_1 \wedge \bar{P}_1) \wedge (\bar{e}_2 \wedge \bar{P}_2) \wedge \ldots)$$

This follows from the fact that $\bar{P}_j$ is exactly the event that no preceding treewalk counts a leaf in V. By the definition of conditional probability, we know that this is equal to:

$$\Pr(\bar{e}_1 \wedge \bar{P}_1) \cdot \Pr(\bar{e}_2 \wedge \bar{P}_2 | \bar{e}_1 \wedge \bar{P}_1) \cdot \Pr(\bar{e}_3 \wedge \bar{P}_3 | \bar{e}_1 \wedge \bar{P}_1, \bar{e}_2 \wedge \bar{P}_2) \cdot \ldots$$

Finally, from Lemma 5, we know that $\Pr(\bar{e}_i \wedge \bar{P}_i | \bar{e}_1 \wedge \bar{P}_1, \bar{e}_2 \wedge \bar{P}_2, \bar{e}_3 \wedge \bar{P}_3, \ldots, \bar{e}_{i-1} \wedge \bar{P}_{i-1}) \leq 1 - |V|/|U_i|$, as Lemma 5 holds given any arbitrary marked and counted leaves. □

4.4 Phase Analysis

We now show that, in every phase, there are $O(p)$ complete treewalks. We consider the two epochs separately, to obtain the following:

Claim 7 *For every phase i in the first epoch, $|B_i| = O(p + k)$ with probability at least $1 - (1/2)^{p+k}$, for any constant k.*

Proof. Fix a phase i in epoch 1 and fix set B_i to be the first $\alpha p + k$ complete treewalks in phase i that complete at the root. We will show that these treewalks count p new leaves, and hence the phase ends.

Assume, for the sake of contradiction, that the number of new leaves counted at the root is $< p$. In this case, there exists a set V of size $|U_i| - p$ such that no leaf in V is counted at the root by the end of this phase. (Note this is only true in epoch 1, where $|U_i| \geq 2p$.) By Lemma 6, and taking a union bound over all possible sets V, the probability that this event occurs is at most:

$$\binom{|U_i|}{p} \cdot \left(\frac{p}{|U_i|}\right)^{\alpha p + k} \leq e^p \cdot \left(\frac{p}{|U_i|}\right)^{(\alpha - 1)p + k} \leq \left(\frac{1}{2}\right)^{p+k}, \text{ for } \alpha \geq 4.$$

□

Claim 8 *For every phase i in the second epoch, $|B_i| = O(p + k)$ with probability at least $1 - (1/2)^{p+k}$, for any constant k.*

Proof. Fix a phase i in epoch 2 and fix set B_i to be the first $\alpha p + k$ complete treewalks in phase i that complete at the root. We will show that these treewalks count $|U_i|/2$ new leaves (halving the number of remaining leaves), and hence the phase ends.

Assume, for the sake of contradiction, that the number of new leaves counted at the root is $< |U_i|/2$. In this case, there exists a set V of size $|U_i|/2$ such that no leaf in V is counted at the root by the end of this phase. By Lemma 6, and taking a union bound over all possible sets V, the probability that this event occurs is at most:

$$\binom{|U_i|}{|U_i|/2} \cdot \left(\frac{1}{2}\right)^{\alpha p + k} \leq e^p \cdot (2)^{|U_i|/2} \cdot \left(\frac{1}{2}\right)^{\alpha p + k} \leq e^{2p} \cdot (1/2)^{\alpha p + k} \leq \left(\frac{1}{2}\right)^{p+k}, \text{ for } \alpha \geq 6.$$

□

Summing up, we have obtained the following.

Lemma 9 *For every phase i and for any constant k, $|B_i| = O(p+k)$ with probability at least $1-(1/2)^{p+k}$.*

We classify phases as either *failed* or *successful*. A phase is successful if it contains at most αp operations, and is failed otherwise. Claims 7 and 8 show that the probability that a phase is failed is at most $(1/2)^p$. We now bound the amount of extra treewalk operations (after the first αp) that are scheduled in failed phases. We call these extra treewalk operations *wasted*. The expected number of extra treewalks for a phase i in either epoch is at most $\sum_{k=1}^{\infty} k/2^{p+k} = 2/2^p = 1/2^{p-1}$. Therefore, the total amount of wasted work in all phases of the execution is, in expectation, at most $(m/p + \log p)/2^{p-1}$.

Therefore, by a Chernoff bound, the probability that the total number of wasted operations is greater equal $4(m + p\log p)$ is at most

$$(1/e)^{p2^{p+1}(m/p+\log p)/2^{p+1}} = (1/e)^{m+p\log p} \leq (1/e)^{m+p}.$$

Summing up, we have showed that, with probability at least $1-(1/e)^{m+p}$, the total number of treewalk operations performed during the execution is at most $(\alpha+6)(m+p\log p) = O(m+p\log p)$ (where the extra factor of two comes from the at most p operations that might execute across distinct phases). Observing that each treewalk performs at most one task, and each treewalk costs $O(\log^2 m)$ steps, we conclude:

Theorem 10 *With probability at least $(1-1/e^{m+p})$, the total number of tasks executed during an execution is $O(m + p\log p)$, and the total number of steps taken is $O(m\log^2 m + p\log p\log^2 m)$.*

By assigning $\log^2 m$ tasks to each leaf, we achieve the following:

Theorem 11 *The total number of tasks executed during an execution and the total number of steps taken is $O(m + p\log p\log^2 m)$, with probability at least $(1 - 1/e^{m/\log^2 m+p})$.*

5 Analysis of Deterministic To-Do Trees

In this section, we show (via the probabilistic method) that there exists a deterministic To-Do Tree algorithm. In our approach, we assume that each process determines its sequence of random bits prior to the execution of the algorithm; the randomized To-Do Tree algorithm then proceeds as before using the predetermined randomness. Each treewalk uses a fixed number of random bits, regardless of whether it gets stuck in the tree or arrives at a leaf. (It discards any unused random bits.) We show that, with high probability, the algorithm still achieves good performance, implying that there exists a good initial sequence of (deterministic) bits.

The added difficulty is that the adversary now gets to see the entire sequence of random choices in advance. Thus, it can schedule the processes based on their future random choices, attempting to prevent them from making progress.

5.1 Overview

Our primary goal is to bound the number of treewalks that processes execute. As the execution progresses, leaves get *marked* and then *counted* at the root. The algorithm terminates when there are no *available* leaves. (See the next section for precise definitions.)

We divide the execution into phases such that $\Theta(1/\log m)$ of the remaining leaves are completed in each phase. The key technical result is a bound on the number of treewalks that take place in each phase (Lemma 18 and Lemma 19). The adversary, by scheduling treewalks, determines the length of each phase. We fix a particular assignment of treewalks to phases, and show that the probability of a phase being too long is exponentially small. The phase length must be long enough (containing at least $\Theta(p\log^3 m)$ treewalks)

so as to get sufficiently small probabilities, but short enough that we can ensure a sufficient percentage of treewalks in that phase succeed.

Within a phase, we analyze the leaves selected by the treewalks. Since the random numbers are fixed in advance, and the adversary can influence the treewalks by controlling their scheduling, it is difficult to determine at which leaf a given treewalk will land. On the other hand, since the random choices are fixed, we can analyze the *a priori* distribution of the treewalks, independent of the actual scheduled execution. As such, treewalks choose tasks uniformly at random from the set of remaining tasks (Claim 13), and the treewalks are "well spread out" in the space of available leaves (Claim 15). We can also show that if the treewalks do not encounter too much disruption in the tree, they will arrive at their "targeted" leaf (Claim 14).

We then analyze the behavior of these treewalks in the execution. We first bound the extent to which the adversary can distort the treewalks (Claim 17). The adversary may "control" a logarithmic factor more leaves than treewalks in a phase. We bound the impact of this control, and hence the number of treewalks that can be disrupted (Claim 16). This yields the desired bound on the number of treewalks in each phase.

Finally, we enumerate the total number of ways that the adversary can assign treewalks to phases, and take a union bound over all possibilities, yielding our final result.

5.2 Preliminaries

We now analyze the performance of the To-Do Tree algorithm, where all the random choices are made in advance. We prove that there exists an input, i.e. a set of p binary strings, one for each process, such that the total work required to perform m work units is $O(m + p\log^5 m \log^2 \max(m,p))$. Throughout, when we talk about the probability of a certain event, this is in reference to the random choices made before the execution began.

We consider an arbitrary execution of the algorithm, which we split into *phases*. In turn, phases will be grouped into two epochs, as described below. We identify leaves of the tree with tasks. A leaf is *marked* if its associated tasks have been performed, otherwise it is *available*, or *unmarked*.

Fix constants c and $d \geq 1$. The first epoch continues as long as there are at least $cp\log^3 m$ leaves available, for some constant c; each phase in the first epoch completes at least a $1/d\log m$ fraction of the remaining available leaves, where $d \geq 1$ is a constant. The second epoch starts when there are less than $cp\log^3 m$ leaves counted at the root, and each phase completes a $1/d\log m$ fraction of the remaining available leaves.

More precisely: For $i \geq 1$, let s_i be the count at the root min-register at the beginning of phase i. Phase 1 starts with the first operation (i.e., $s_1 = m$), and ends with the first read or write operation on the root min register that returns a value $\leq m(1 - 1/d\log m)$. For $i \geq 2$, phase i starts as soon as phase $i-1$ ends, and finishes with the first read or write operation on the root max register that returns a value $\leq s_i(1 - 1/d\log m)$. Epoch one contains all phases with the property that $s_i \geq cp\log^3 m$, where c is a constant.

Epoch two starts with the first phase in which $s_i < cp\log^3 m$. We split phases in the same way: phase i ends with the first read or write operation on the root max register that returns a value $\leq s_i(1-1/(d\log m))$, for d constant.

Claim 12 *Epoch 1 has $O(\log^2 m)$ phases, and epoch 2 has $O(\log m \log p + \log m \log\log m)$ phases.*

We say that phase i in the first epoch is *successful* if the number of complete walks scheduled in i is $O(\max(s_i/\log m, p\log^3 m))$. Otherwise, the phase is *failed*. A phase i in the second epoch is *successful* if the number of complete walks scheduled in i is $O(p\log^3 m)$; otherwise, the phase is *failed*.

For much of the remainder of the proof, we now fix a specific schedule, determined by the adversary. That is, the adversary chooses which processes take steps in which order, which determines the way in

which the execution is partitioned in phases. We will show that for each specific schedule, we can bound the number of treewalks with very high probability. At the end, we will take a union bound over all the (exponentially many) possible schedules.

5.3 The First Epoch

For phase i, let $k = \Theta(\max(s_i/\log m, p\log^3 m))$. Let B_i be the set of complete walks in phase i. Define the reference tree RT_i as in Section 4, and let U_i be the set of leaves unmarked in the reference tree at the beginning of phase i. Note that $|U_i|$ is at least s_i, by definition. The goal of this section is to show that, with very high probability, phase i completes after k complete treewalks. That is, k complete treewalks are sufficient to count at least $s_i/d\log m$ new leaves at the root.

We fix a set B_i of $k = \alpha\max(s_i/\log m, p\log^3 m)$ complete treewalk operations during phase i, chosen to be scheduled by the adversary, for some constant α.

For a given treewalk w, we define the target of w to be the leaf that is reached via a treewalk in the (unmodified) reference tree. Each such treewalk chooses uniformly from the available leaves:

Claim 13 *The probability that a walk $w \in B_i$ targets a specific leaf (in the reference tree) is $1/s_i$.*

Proof. We prove the claim by induction on the height of the subtree at which the walk is located. Recall that we perform the walk and decide the target in the reference tree at the beginning of the phase, therefore there is no concurrency, and the counts at the nodes do not change from their initial configuration.

The induction statement is that if the walk w is currently located at a subtree rooted at v with s available leaves, the probability that it hits a particular descendant available leaf is $1/s$. The base case, when the walk is located at a leaf, is trivial.

For the induction step, assume the walk is located at node v, with s descendant leaves. Pick a descendant leaf ℓ, and let x and y be the min register counts in the reference tree corresponding to the left and right children of v, respectively. Without loss of generality, assume that ℓ is in the left subtree at v.

The probability that w hits ℓ is the probability that it goes in the direction of ℓ's subtree at node v, times the probability that it hits ℓ starting from node x. Using the induction step at the left child, the resulting expression is

$$\frac{x}{x+y} \cdot \frac{1}{x} = \frac{1}{x+y} = \frac{1}{s},$$

which completes the induction. The proof of the main claim follows by applying the induction at the root node. □

The muting threshold. For every node v in the reference tree, we define the *muting threshold* for v as follows: Let j be the value of the min-counter at v in the reference tree for this phase; then the muting threshold $t_v = j(1 - 1/(\beta\log m))$, for some constant $\beta \geq 1$. In the real tree, we say that v is *muted* from the first step after which all but t_v of the tasks in v's sub-tree are counted at node v.

Similarly, we say that a subtree is muted if its root node is muted. The intuition is that in muted subtrees, counter values may be quite different from in the reference tree, and hence, we make no assumptions on which leaves are reached by walks entering a muted subtree. Any subtree contained in a muted subtree is also muted.

Live tree walks. We have already fixed a set of treewalks $w_1, w_2, \ldots$ that complete in phase i (by assumption). We have defined the *target* of a treewalk w to be the leaf that the treewalk would reach in the descent phase if it were executed in the (unmodified) reference tree. We define an *adversarial reference tree* to be a copy of the reference tree in which the adversary is allowed to decrement any of the counters in the tree, as

long as no min-register at node v falls below the muting threshold t_v. We say that a treewalk w is *alive* if it reaches its target in every possible adversarial reference tree. If a treewalk is not alive, then it is *dead*.

We now prove that at least a constant fraction of the k walks in B_i remain alive.

Claim 14 *Let S be the set of live walks during this phase. With probability at least $1-(1/e)^{k/16}$, $|S| > k/2$.*

Proof. We first show that the probability that a given walk w is dead is at most $1-(1/e)^{1/\beta}$. Consider some step where the treewalk w is at some node v. It reads x from the min-register at the left child and y from the min-register at the right child. Assume (without loss of generality) that the target path from v goes to the left. Then the worst-case adversarial reference tree adopts the minimum value for x, i.e., $x' = x(1-1/\beta\log m)$. Thus, the probability that treewalk w remains alive is $x'/(x'+y) \geq x'/(x+y)$, and the probability that treewalk w is dead is $\leq (x/(x+y))(1/\beta\log m) \leq (1/\beta\log m)$. Thus the probability that w remains alive for all $\log m$ steps is $\geq (1-1/\beta\log m)^{\log m} = e^{-1/\beta}$.

For β large enough, the probability that a walk is live is $> 3/4$. Therefore, by a Chernoff bound, the probability that less than $k/2$ of the walks in this phase reach their targets is at most $(1/2)^{k/16}$. Let S be the set of live walks. With very high probability, $|S| > k/2$. □

Sparse tree walks. Let U_i be the set of unmarked leaves at the beginning of this phase. Let $T = \{t_1, t_2, \ldots\}$ be the leaves targeted by the walks $B_i = \{w_1, w_2, \ldots\}$ scheduled in this phase, respectively, and let L be a list of these targeted leaves, ordered from left to right.

Let B be an arbitrary set of walks from B_i, and let $T(B)$ be the set of target leaves for walks in B. We say that a walk $w \in B$ targeting leaf $t \in T(B)$ is *sparse* if *none* of the $(\log m)/\alpha - 1$ predecessors of leaf t in the list L are in $T(B)$ *and* none of the $(\log m)/\alpha - 1$ successors of leaf t in the list L are in $T(B)$. Intuitively, if all the leaves in B are sparse, then they are well spaced out in the list of leaves.

We now argue that, with very high probability, there exists a set $V \subseteq S$ of at least $\alpha s_i/16\log m$ live treewalk operations scheduled in this phase that are all *sparse*.

Claim 15 *With probability at least $1-(1/e)^{k/16}$, there exists a set V of $\alpha s_i/(16\log m)$ sparse live walks.*

Proof. Let L be the list of leaves, ordered from left-to-right, that are unmarked in the reference tree in phase i. We partition the list L of available leaves into $\alpha s_i/\log m$ *bins*, where each bin contains $(\log m)/\alpha$ leaves consecutive in the list L.

We then consider a set S of live walks of size $\alpha s_i/2\log m$, and bound the probability that a bin contains exactly one leaf targeted by a walk in S. Such a set S exists as per Claim 14.

Notice that the target of a treewalk w is independent of whether the treewalk is live or dead. Since each treewalk targets a leaf chosen uniformly at random, as per Claim 13, we can conclude that each treewalk in S is assigned to a bin at random.

Since there are $\alpha s_i/\log m$ bins and $\alpha s_i/2\log m$ treewalks, the expected number of bins with exactly 1 ball is $\alpha s_i/2\log m \cdot e^{-1/2} \geq \alpha s_i/4\log m$. By a Chernoff bound, we obtain that the probability that less than $\alpha s_i/8\log m$ of the bins have exactly one ball in them is at most $(1/e)^{\alpha s_i/16\log m}$. We call this the set of singleton bins.

Let V be the set consisting of the live treewalks that choose every alternate singleton bin, ensuring that there are at least $\log m/\alpha - 1$ leaves between every pair of leaves chosen by treewalks in V. Since $|V| \geq \alpha s_i/16\log m$, we are done if $k/16 \leq \alpha s_i/16\log m$ (i.e., the error probability is as desired).

Consider the case where $k = \alpha p\log^3 m$, which may be larger than $s_i/\log m$ in phases towards the end of the first epoch. We split the k walks into batches of $\alpha s_i/\log m$. Notice that there are $k\log m/\alpha s_i$ such batches. We repeat the procedure above for each batch. Since the trials are independent, the probability that no batch contains a set of $\alpha s_i/16\log m$ sparse walks is at most

$$(1/e)^{(\alpha s_i/16\log m)\cdot(k\log m/\alpha s_i)} \leq (1/e)^{k/16}.$$

□

Therefore, we have obtained that there exists a set V of at least $\alpha s_i/16 \log m$ walks that are all sparse, and all reach their targets, with probability at most $(1/e)^{k/16}$. Notice that none of these walks are diverted at nodes that are not muted, as they are all live. The only option left for the adversary, to prevent the phase from succeeding, is to mute the subtrees that many of these walks access, thus ensuring that the total amount of work that these walks perform is small.

Analyzing the real tree. Consider a treewalk w in the set V. We now examine what happens when it executes in the real tree. Recall that it is complete, meaning that it proceeds down the tree and returns to the root. There are two possibilities: (i) the treewalk continues to its target, or (ii) some node encountered by the tree walk is muted by the end of the tree walk. Since the treewalk is on target, these are the only two possibilities. If no counter that it reads exceeds the muting threshold, then, since by assumption the treewalk is on target, it proceeds to its target as it would in the adversarial reference tree. (Notice that registers may differ from their values in the reference tree for many reasons: due to other concurrent treewalks, or due to treewalks delayed from previous phases.)

Success point. We define the success point to be the first step after which at least $s_i/d \log m$ new leaves are counted at the root, i.e., there are at most $s_i(1 - 1/d \log m)$ incomplete tasks remaining. Consider the subset of treewalks in V that complete prior to the success point. As already discussed, some subset of these treewalks arrive at their target; the rest traverse a muted node. In the latter case, we can conclude that their target was a descendent of a muted node.

Shadow. Given any muted subtree, we say that all its unmarked leaves are *in shadow*. Thus, the set Sh of all leaves in shadow is the union of the sets of unmarked leaves of all muted subtrees, as decided by the adversary. Notice that if the target leaf t of a walk w is in Sh, then the walk will enter a muted subtree along its path to the target leaf, and may be diverted arbitrarily inside this subtree. We define a *contiguous segment* g in Sh as a maximal set of consecutive leaves from the list L with the property that all leaves are in Sh. We partition the shadow set Sh in contiguous segments $G = \{g_1, g_2, \ldots\}$.

Let us examine a segment g from the point of view of walks in the set V of $\alpha s_i/16 \log m$ sparse on-target walks. Assume that some of the walks in V target leaves in the segment g. Then they will enter the respective muted subtrees, and may be arbitrarily diverted towards other leaves. In the best case for the adversary, they all get diverted to the same leaf. However, notice that if at least one walk w targets a leaf in the segment, then at least one leaf in the segment gets counted at the root by the end of the phase. Second, more importantly, if some subset H of walks in V target leaves in g, then the size of the segment g has to be at least $\log m(|H| - 1)/\alpha$. We prove this as follows.

Claim 16 *Given a contiguous segment g in shadow, and a non-empty set H of treewalks in V that target leaves in g, then: i) at least one leaf in the segment gets counted at the root in this phase and ii) the size of the segment g has to be at least* $\log m(|H| - 1)/\alpha$.

Proof. For the first statement, notice that there exists at least one walk w which targets a leaf in g. At some point during this phase, this walk enters one of the muted subtrees whose leaves form the segment g. Let v be the root node of this subtree. Since the subtree is muted, at least one of the counters at the children of v is less than its value in the reference tree. By Lemma 2, we obtain that there exists a leaf ℓ in $U_i \cap g$ counted at that subtree. Since, by assumption, the walk w reaches the root in this phase, the leaf ℓ will also be counted at the root by the end of this phase.

For the second statement, let $H = h_1, h_2, \ldots$ be the set of walks in V that target leaves in g. Without loss of generality, let h_1 be the walk targeting the leftmost leaf in g. Since they are *sparse*, by definition, the rest of the walks in $h_2, h_3, \ldots$ must target distinct leaves $\ell_2, \ell_3, \ldots$ that are *each* spaced at at least $(\log m)/\alpha - 1$ leaves from the previous leaf. Hence, by a simple counting argument, the size of the segment g has to be at least $\log m(|H| - 1)/\alpha$. □

We now calculate the total number of leaves that can be in shadow, prior to the success point: there can be at most $2\beta s_i/(d \log m)$ such leaves.

Claim 17 *For $d > 2\beta$ constant, at most $2\beta s_i/d$ leaves are in shadow during phase i of epoch 1.*

Proof. Consider an arbitrary muted subtree ST, and let ℓ be the number of available leaves in ST at the beginning of the phase. The subtree has been muted because at least $\ell/(\beta \log m)$ of its available descendant leaves are marked during this phase. Therefore, ℓ is at most $(\beta \log m)$ times larger than the number of leaves marked during this phase in ST. This holds for any muted subtree. Therefore, in sum, the total available leaves in muted subtrees is at most $(\beta \log m)$ times the number of marked leaves in muted subtrees.

On the other hand, the total number of marked leaves in muted subtrees (for this phase) is at most the number of marked leaves in this phase, which, since we have not yet reached the success point, is at most $s_i/(d \log m) + p \leq 2s_i/(d \log m)$. By the previous argument, the total number of leaves in muted subtrees is less than $2\beta \log m \cdot s_i/(d \log m) = 2\beta s_i/d$. □

At this point, we can assemble all the pieces and prove the main lemma of this section, which shows that each phase i in epoch 1 contains only k complete treewalks, with very high probability:

Lemma 18 *Let $\alpha \geq 1$ be a constant. For $i \geq 1$, the probability that $k = \alpha \max(s_i/\log m, p \log^3 m)$ complete treewalk operations in phase i count* less *than $s_i/d \log m$ leaves at the root is at most $(1/e)^{k/16}$.*

Proof. Recall that we have identified a set V of at least $\alpha s_i/16 \log m$ walks which are sparse and on target, with probability at least $1-(1/e)^{k/16}$ (Claim 15). These walks are partitioned across segments $g_1, g_2, \ldots$, as decided by the adversary. For each segment, one walk that targets the segment is guaranteed to perform one unit of work, whereas the work performed by the rest could be wasted (since all walks may reach the same leaf). For each segment g, we pick a walk h to be the useful walk, and call the other walks *wasted*. However, Claim 16 ensures that for each wasted walk in V reaching a segment g, the segment g has to contain at least $\log m/\alpha$ extra leaves. In turn, based on a simple counting argument, Claim 17 implies that the number of wasted walks in V is at most

$$\frac{2\beta s_i}{d} \cdot \frac{\alpha}{\log m} = \frac{2\alpha\beta s_i}{d \log m}.$$

Therefore, the number of walks in V that are either not targeting leaves in shadow or are *useful* is at least

$$\frac{\alpha s_i}{16 \log m} - \frac{2\alpha\beta s_i}{d \log m} \geq \frac{s_i}{d \log m},$$

where we have picked $d = \alpha$ and $\alpha \geq 16(2\beta + 1)$. This occurs with probability at least $1-(1/e)^{k/16}$. Since no two such walks may hit the same leaf, it follows that, for sufficiently large α, the main claim follows. □

5.4 The Second Epoch

The second epoch starts with at most $cp \log^3 m$ leaves not counted at the root of the tree, where c is a costant. For each phase i in the second epoch, assume $\Theta(p \log^3 m)$ complete walks. The reference tree has s_i leaves not counted at the root at the beginning of the phase. We now bound the failure probability for a phase i in the second epoch.

Lemma 19 *Let $\alpha, c, d \geq 1$ be constants. For $i \geq 1$, the probability that $\alpha p \log^3 m$ complete treewalk operations in phase i count* less *than $s_i/(d \log m)$ leaves at the root is at most $(1/e)^{(\alpha-c)p \log^3 m}$.*

Proof. We fix a set of $k = \alpha p \log^3 m$ complete treewalk operations, chosen to be scheduled by the adversary.

The muting threshold. For every node v in the reference tree, we define the *muting threshold* for v as follows: Let j be the value of the min-counter at v in the reference tree for this phase; then the muting threshold $t_v = j(1 - 1/(\beta \log m))$, for some constant $\beta \geq 1$. In the real tree, we say that v is *muted* from the first step after which all but t_v of the tasks in v's sub-tree are counted at node v.

Live tree walks. We have already fixed a set of treewalks $w_1, w_2, \ldots$ that complete (by assumption). We define the *target* of a treeewalk w to be the leaf that the treewalk would reach in the descent phase if it were executed in the (unmodified) reference tree. We define an *adversarial reference tree* to be a copy of the reference tree in which the adversary is allowed to decrement any of the counters in the tree, as long as no min-register at node v falls below the muting threshold t_v. We say that a treewalk w is *alive* if it reaches its target in every possible adversarial reference tree. If a treewalk is not alive, then it is *dead*.

Analysis of tree walks. We now argue that, with very high probability, there exists a set of at least $s_i/4$ tasks such that every task is targeted by a live tree walk. Notice that whether or not one treewalk is alive and hits its target is independent of whether another treewalk is alive and hits its target.

Let V be a set of $s_i/4$ available leaves. Fix a given treewalk w. By Claim 13, the probability that w targets a leaf in V is $1/4$.

We now calculate the probability that treewalk w is dead. Consider some step where the treewalk is at some node v. It reads x from the min-register at the left child and y from the min-register at the right child. Assume (without loss of generality) that the target path from v goes to the left. Then the worst-case adversarial reference tree adopts the minimum value for x, i.e., $x' = x(1-1/\beta \log m)$. Thus, the probability that treewalk w remains alive is $x'/(x'+y) \geq x'/(x+y)$, and the probability that treewalk w is dead is $\leq (x/(x+y))(1/\beta \log m) \leq (1/\beta \log m)$. Thus the probability that w remains alive for all $\log m$ steps is $\geq (1 - 1/\beta \log m)^{\log m} = e^{-1/\beta}$. Thus, for sufficiently large choice of β, the probability that w is dead is $< 1/4$.

Thus by a union bound, the probability that treewalk w either hits V or dies is $< 1/2$. Since each treewalk is independent, the probability that all k treewalks either hit V or die is $\leq 1/2^k$.

Notice that there are at most $\binom{s_i}{s_i/4} \leq (4e)^{s_i/4} \leq 2^{s_i}$ possible sets V. Since we are in the second epoch, the probability that every live treewalk hits a set of size $s_i/4$ is at most

$$2^{s_i}/2^{\alpha p \log^3 m} \leq \left(\frac{1}{2}\right)^{(\alpha - c) p \log^3 m}.$$

From this, since we pick $\alpha \geq c+1$, we conclude that, with very high probability, the set of *live* treewalks targets at least $s_i/4$ of the leaves.

Analyzing the real tree. Consider a live treewalk w. We now examine what happens when it executes in the real tree. Recall that it is complete, meaning that it proceeds down the tree and returns to the root. There are two possibilities: (i) the treewalk continues to its target, or (ii) some node encountered by the tree walk is muted by the end of the tree walk. Since the treewalk is alive, these are the only two possibilities. If no counter that it reads exceeds the muting threshold, then, since by assumption the treewalk is alive, it proceeds to its target as it would in the adversarial reference tree. (Notice that registers may differ from their values in the reference tree for many reasons: due to other concurrent treewalks, or due to treewalks delayed from previous phases.)

Success point. Fix a constant $d \geq 1$. We define the success point to be the first step after which at least $s_i/(d \log m)$ new leaves are counted at the root, i.e., there are at most $s_i(1 - 1/(d \log m))$ incomplete tasks remaining. Consider the set of live treewalks that complete prior to the success point. As already discussed, some subset of these live treewalks arrive at their target; the rest traverse a muted node. In the latter case, we can conclude that their target was a descendent of a muted node.

We now calculate the total number of leaves that can be descendants of muted nodes, prior to the success point. In fact, there can be at most s_i/d such leaves.

Claim 20 *At most $\beta s_i/d$ leaves may be descendants of muted nodes.*

Proof. Consider an arbitrary muted subtree ST, and let ℓ be its number of available leaves. The subtree has been muted because at least $\ell/\beta \log m$ of its available descendant leaves are marked during this phase. Therefore, ℓ is at most $\beta \log m$ times larger than the number of leaves marked during this phase in ST. This holds for any muted subtree. Therefore, in sum, the total available leaves in muted subtrees is at most $\beta \log m$ times the number of marked leaves in muted subtrees.

On the other hand, the total number of marked leaves in muted subtrees (for this phase) is at most the number of marked leaves in this phase, which, since we have not yet reached the success point, is at most $s_i/(d \log m)$. By the previous argument, the total number of leaves in muted subtrees is less than $\beta \log m \cdot s_i/(d \log m) = \beta s_i/d$. $\square$

Finally, recall we have already shown that, with very high probability, the live treewalks target at least $s_i/4$ of the leaves. At most $\beta s_i/d$ of these leaves can be descendants of muted nodes. Therefore, either $s_i/4 - \beta s_i/d$ of the leaves are reached by live treewalks, or the execution reaches a success point. However, in the first case, for $d > 8\beta$, at least $s_i/8$ of the leaves are reached by live treewalks. For $d \geq 8$, we are done. The probability of failure is at most: $(1/2)^{(\alpha-c)p \log^3 m}$.

5.5 Counting Schedules

We have upper bounded the probability of failure for each phase in the two epochs, under the assumption that the schedule is fixed by the adversary. The probability of failure is so small that we can take a union bound over all the possible schedules and conclude the proof.

Each treewalk uses a fixed number of random numbers, and hence the number of schedules is determined by the number of different treewalk interleavings that the adversary can generate during an execution of the algorithm.

To model this, we assume that each process has a (possibly infinite) sequence of walks at its disposal. The adversary chooses how many walks to schedule for each process, and their interleaving among processors. (For each process, the adversary can only schedule walks in order.)

For example, in the first phase of epoch one, the adversary chooses to schedule $\alpha m/\log m$ complete walks, along with up to p incomplete walks that may start during this phase at the p distinct processes. These treewalks can be allocated to any subset of the processes in any quantity. The number of possible combinations is upper bounded by

$$\binom{\alpha m/\log m + 2p}{\alpha m/\log m} = \binom{\alpha m/\log m + 2p}{2p} \leq 2^{c' p \log m}.$$

Let $j_1 = O(\log^2 m)$ be the number of phases that epoch one lasts for. Since any interleaving of the first j_1 phases is a concatenation of interleavings that may occur in each phase, the number of possible interleavings of the j_1 phases in epoch one is at most

$$\prod_{i=1}^{j_1} \binom{\alpha s_i/\log m + 2p}{2p}$$

Similarly, if epoch two lasts for $j_2 = O(\log p \log m)$ phases, the number of possible interleavings in the entire execution is at most

$$\prod_{i=1}^{j_1} \binom{\alpha s_i / \log m + 2p}{2p} \prod_{i=1}^{j_2} \binom{\alpha p \log^3 m + 2p}{2p} \leq (2^{c' p \log m})^{j_1 + j_2} \leq 2^{cp \log m (\log^2 m + \log p)}$$

We now split the exposition into two cases, depending on the relation between $\log p$ and $\log^2 m$.

The case $p < m^{\log m}$. If $\log p < \log^2 m$, then, by Lemma 18 and Lemma 19, there exists a constant $\alpha \geq 1$ such that, for a given scheduling of treewalks, the probability that any of the $O(\log^2 m + \log p) = O(\log^2 m)$ phases fails is at most

$$(j_1 + j_2) \cdot \left(\frac{1}{2}\right)^{\alpha p \log^3 m} \leq \left(\frac{1}{2}\right)^{(\alpha - 1) p \log^3 m}.$$

Finally, by a union bound over all possible schedules, the probability that there exists exists an interleaving of treewalks for which some phase fails is at most

$$\left(\frac{1}{2}\right)^{(\alpha - c - 1) p \log^3 m} < 1$$

for $\alpha > c + 1$. Therefore, there exists a sequence of random bits for each process for which all phases are *successful*.

The case $p \geq m^{\log m}$. Otherwise, the number of interleavings is at most

$$2^{cp \log p \log m}.$$

In this case, notice that the first epoch does not exist, and we start immediately with the second epoch, in which phases are defined identically, but are defined to be *successful* if they contain $\Theta(p \log p \log m)$ complete treewalks. By an identical proof to that of Lemma 19, we obtain that each phase contains $\Theta(p \log p \log m)$ with probability at least $1 - (1/2)^{\Theta(p \log p \log m)}$.

Recall that the total number of interleavings in this case is $2^{cp \log p \log m}$. Again, for an appropriate choice of constants $\alpha > c + 1$, we can offset this by the error probability over all phases. The probability that there exists an interleaving of treewalks for which some phase fails is at most

$$\left(\frac{1}{2}\right)^{(\alpha - c - 1) p \log p \log m}.$$

Therefore, there exists a sequence of random bits for each process for which all phases are successful.

5.6 Total Executed Tasks and Total Work

The last step is to upper bound the total work performed during an execution in which all phases are successful.

For this, notice that, as long as the number of available leaves s_i at the beginning of the phase verifies $s_i / \log m \geq p \log^3 m$, we are "spending" $\Theta(s_i / \log m)$ treewalk operations to perform $\Theta(s_i / \log m)$ work, thereby the work ratio per phase is (asymptotically) optimal. After this point, we are spending $\Theta(p \log^3 m)$ treewalk operations to mark $s_i / \log m$ new leaves, if $m^{\log m} \geq p$, and $\Theta(p \log p \log m)$ treewalk operations to mark the same number of leaves, otherwise. Therefore, the total number of treewalk operations we employ can be upper bounded by:

$$\alpha[(m - p \log^4 p) + (p \log^2 m \log \max(m, p)) \cdot \log m (\log(p \log^3 m))] = \\ O(m + p \log \max(m, p) \log^3 m (\log p + \log \log m)).$$

Since each treewalk may execute a single task, this bounds the total tasks executed by the deterministic algorithm. To bound the total amount of *work*, i.e. shared-memory operations, recall that each treewalk operation has cost $O(\log^2 m)$. Thus, this version of the algorithm verifies the following.

Theorem 21 *There exists a deterministic To-Do Tree algorithm executing* $O(m + p\log^3 m(\log p + \log\log m)\log\max(m,p))$ *tasks, with work* $O(m\log^2 m + p\log^5 m(\log p + \log\log m)\log\max(m,p))$.

By assigning $\log^2 m$ tasks to each leaf in the tree, we obtain that the total amount of work is

$$\alpha(m/\log^2 m + p\log^3 m(\log p + \log\log m)\log\max(m,p))(2\log^2 m) = O(m + p\log^5 m(\log p + \log\log m)\log\max(m,p)).$$

The resulting algorithm verifies the following.

Theorem 22 *For general m and p, there exists a deterministic To-Do Tree algorithm with total work and number of tasks executed* $O(m + p\log^5 m(\log p + \log\log m)\log\max(m,p))$.

6 Conclusion

We present randomized and deterministic algorithms for the shared-memory task allocation problem. Our algorithms are efficient for general values of m and p, and match the $\Omega(m + p\log p)$ lower bound of [13] to within logarithmic factors.

We have not reached the limits of our techniques. We believe that our approach can be refined to improve the deterministic bounds. We conjecture that there exists a tighter analysis, showing that the deterministic To-Do Tree can have the same asymptotic work and tasks-executed bounds as the randomized To-Do Tree. Although we omit details here, our approach may also be used to analyze other variants of asynchronous task allocation, such as *collect* [2], in which processes need to aggregate register values and *do-most* [19], in which only a fraction of the tasks need be performed.

A Background on Max/Min-Registers

We recall here the max-register construction of Aspnes et al. [4]. The min-registers we employ can be implemented in the same manner as a max-register, where every value read and written is simply *subtracted* from the default initial value.

Intuitively, a max-register is a data structure that holds the highest value ever written to it, and defines a default maximal value v_{max} which it may store. We construct the max-register recursively. If the maximal value $v_{\mathsf{max}} = 1$, then the max-register consists of a single bit b. Bit b is initialized to 0 and set to 1 the first time there is a non-zero write.

```
1  max-register⟨1⟩
2     b : bit, initially 0
3     procedure write(v)
4        if v = 1 then b ← 1
5     procedure read()
6        return b
```

In general, for $v_{\mathsf{max}} > 1$, the max-register consists of two max-registers $M1$ and $M2$ of size approximately $v_{\mathsf{max}}/2$, and a binary bit b. If the value v being written is less than $v_{\mathsf{max}}/2$, then it is written to max-register $M1$; otherwise, the value $v - v_{\mathsf{max}}/2$ is written to max-register $M2$, and the binary bit b is updated to 1.

```
1  max-register⟨v_max⟩
2     b : bit, initially 0
3     M1 : max-register⟨⌊v_max/2⌋⟩
4     M2 : max-register⟨⌈v_max/2⌉⟩
5     procedure write(v)
6        if v ≤ ⌊v_max/2⌋ then  M1.write(v)
7        else
8              M2.write(v − ⌊v_max/2⌋)
9              b ← 1
10    procedure read()
11       if b = 0 then return M1.read()
12       else return M2.read() + ⌊v_max/2⌋
```

We assume that the max-register has initial value 0. This construction has the following complexity.

Theorem 23 *(Aspnes et al. [4]) There exists a linearizable, wait-free max-register construction against an adaptive adversary where each operation has cost* $O(\log v_{\mathsf{max}})$.

We implement min-registers using this construction, as follows. Each min-register has a default maximal value v_{max}. When writing value v to a min-register, the process simply writes $v_{\mathsf{max}} - v$ to the max register. To obtain the min-register's value, the process reads value v from the max-register and returns $v_{\mathsf{max}} - v$.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[2] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proc. 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 401–411, 1994.

[3] R. J. Anderson and H. Woll. Algorithms for the certified write-all problem. *SIAM J. Comput.*, 26:1277–1283, October 1997.

[4] J. Aspnes, H. Attiya, and K. Censor. Max registers, counters, and monotone circuits. In *Proc. 28th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 36–45, 2009.

[5] H. Attiya and J. Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.

[6] Y. Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *Proc. 16th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 209–218, 1997.

[7] Y. Aumann, Z. M. Kedem, K. V. Palem, and M. O. Rabin. Highly efficient asynchronous execution of large-grained parallel programs. In *Proc. 34th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 271–280, 1993.

[8] Y. Aumann and M. O. Rabin. Clock construction in fully asynchronous parallel systems and pram simulation. *Theoretical Computer Science*, 128(1-2):3–30, June 1994.

[9] A. Aziz, A. Prakash, and V. Ramachandran. A near optimal scheduler for switch-memory-switch routers. In *Proc. 15th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 343–352, 2003.

[10] M. A. Bender and S. Gilbert. Mutual exclusion with $O(\log^2 \log n)$ amortized work. In *Proc. 52nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 728–737, 2011.

[11] M. A. Bender and C. A. Phillips. Scheduling dags on asynchronous processors. In *Proc. 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 35–45, 2007.

[12] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proc. 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 41–51, 1993.

[13] J. F. Buss, P. C. Kanellakis, P. L. Ragde, and A. A. Shvartsman. Parallel algorithms with processor failures and delays. *J. Algorithms*, 20:45–86, January 1996.

[14] B. S. Chlebus, S. Dobrev, D. R. Kowalski, G. Malewicz, A. Shvartsman, and I. Vrto. Towards practical deteministic write-all algorithms. In *Proc. 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 271–280, 2001.

[15] B. S. Chlebus and D. R. Kowalski. Cooperative asynchronous update of shared memory. In *Proc. 37th Annual ACM Symposium on Theory of Computing (STOC)*, pages 733–739, 2005.

[16] C. Georgiou and A. A. Shvartsman. *Do-All Computing in Distributed Systems: Cooperation in the Presence of Adversity*. Springer, 2008.

[17] J. F. Groote, W. H. Hesselink, S. Mauw, and R. Vermeulen. An algorithm for the asynchronous write-all problem based on process collision. *Distrib. Comput.*, 14(2):75–81, Apr. 2001.

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

[19] Z. M. Kedem, K. V. Palem, M. O. Rabin, and A. Raghunathan. Efficient program transformation for resilient parallel computation via randomization. In *Proc. 24th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 306–317, May 1992.

[20] Z. M. Kedem, K. V. Palem, and P. G. Spirakis. Efficient robust parallel computations. In *Proc. 22rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 138–148, May 1990.

[21] S. Kentros, A. Kiayias, N. Nicolaou, and A. A. Shvartsman. At-most-once semantics in asynchronous shared memory. In *Proc. 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 43–44, 2009.

[22] D. R. Kowalski and A. A. Shvartsman. Writing-all deterministically and optimally using a nontrivial number of asynchronous processors. *ACM Trans. Algorithms*, 4:33:1–33:22, July 2008.

[23] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[24] G. Malewicz. A work-optimal deterministic algorithm for the asynchronous certified write-all problem. In *Proc. 22nd Annual Symposium on Principles of Distributed Computing (PODC)*, pages 255–264, 2003.

[25] C. Martel and R. Subramonian. On the complexity of certified write-all algorithms. *J. Algorithms*, 16:361–387, May 1994.

[26] A. Prakash, A. Aziz, and V. Ramachandran. Randomized parallel schedulers for switch-memory-switch routers: Analysis and numerical studies. In *Proc. 23rd Conference of the IEEE Communications Society (INFOCOM)*, 2004.

[27] C. Womach and M. Farach. Randomization, persuasiveness and rigor in proofs. *Synthese*, 134(1-2):71–83, 2003.