2012 IEEE 53rd Annual Symposium on Foundations of Computer Science

How to Allocate Tasks Asynchronously

Dan Alistarh
EPFL
Lausanne, Switzerland
dan.alistarh@epfl.ch

Michael A. Bender
Stony Brook University
and Tokutek, Inc.
New York, USA
bender@cs.stonybrook.edu

Seth Gilbert
NUS
Singapore
seth.gilbert@comp.nus.edu.sg

Rachid Guerraoui
EPFL
Lausanne, Switzerland
rachid.guerraoui@epfl.ch

***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 been studied extensively, both independently 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.

I. Introduction

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

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 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 *write-all*, *do-all*, or *certified do-all*) has been recognized as one of the central problems in distributed computing for decades; the book by Giorgiou and Shvartsman [17] gives a detailed history of this problem. The shared-memory version was introduced over twenty years ago by Kanellakis and Shvartsman [19] 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. [4], [14], [16], [23], [25], [26]. Asynchronous task allocation is also related to *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 [18], mutual exclusion [11], atomic snapshots [1], consensus [7], renaming [13], distributed phase clocks [9], and PRAM simulation [21].

The last twenty years have seen a long-standing quest to establish the complexity of asynchronous task allocation [4], [14], [16], [23], [25], [26]. Various structures such as low-contention permutations [4] and expander-

0272-5428/12 $26.00 © 2012 IEEE
DOI 10.1109/FOCS.2012.41

IEEE computer society

based progress graphs [16] 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 the 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 tasks not yet executed. This strategy is natural and appears periodically in the literature [8], [10], [12], [27]. We build a distributed data structure to perform the selection efficiently, and first analyze its randomized behavior.

We then ask whether this task selection process can proceed without randomization. The main challenge is showing that the randomness can be isolated to the beginning of the execution. In fact, the adversary does get nontrivial extra power from knowing the coin-flip outcomes in advance prior to the execution, but we show that this power is limited. Moving all randomness to the beginning implies, via the probabilistic method, that a deterministic solution exists.

The main results of this paper are the following:

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

The To-Do Tree algorithm improves on existing algorithms in both randomized and deterministic settings; see Table I. In the worst-case, previously 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., [11].

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 [28].

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 $\operatorname{polylog}(m)$) may dominate the scheduling cost; see [8] 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 [22].) The lower bound of [14] still applies here, meaning that number of tasks executed has an $\Omega(m+p\log p)$ lower bound.

Our randomized and deterministic algorithms execute $O(m+p\log p)$ and $O(m+p\log^5 m\log^2\max(m,p))$ tasks, respectively. Thus, our randomized algorithm executes an optimal number of tasks, and our deterministic algorithm executes a number of tasks which is within poly-logarithmic factors of optimal.

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. Thus when there is only one task left to be executed, that task is executed only once all p processes are poised over it; at that point, 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 (i.e., if processes simply choose tasks at random), and $p=\Omega(\log m)$, a process will need $\Omega(m)$ trials to reach this last task. Based on this strategy, the adversary can build an interleaving in which processes require $\Omega(mp)$ 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 randomly whether to turn left or right at every internal node based on the

[1] This example also illustrates the difference in capabilities between the adaptive and oblivious adversaries. This randomized strategy works well 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.

setting	deterministic (general m, p)	deterministic ($p = m$)	randomized (strong adversary)
previous bounds	$O(m + p^{2+\epsilon})$ [23]	$O(m \log^{18} m / \log^3 \log m)$ [16]	$O(m \log m)$ [4]
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	$\Omega(m + p \log p)$ [14]	$\Omega(m \log m)$ [14]	$\Omega(m + p \log p)$ [14]

Table I
SUMMARY OF THE TOTAL WORK BOUNDS AND RELATION TO PREVIOUS WORK.

number of unexecuted descendant leaves of 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 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 the same manner as the shared-memory counter from [5]. Unlike a counter, however, it supports a random-leaf-search operation. The counters at internal nodes of the To-Do Tree are maintained using the “max-register” construction from [5].

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 better 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 (i.e., prior to the beginning of the execution), 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. Moreover, the dependencies among the random choices 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.

Prior 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* [4], [14] 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 [23], 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 [25], using $O(m + p^4 \log m)$ work.

For the special case when $p = m$, Chlebus and Kowalski [16] gave a non-explicit deterministic algorithm with complexity $O(m \log^{18} m / (\log \log m)^3)$ and an explicit variant with complexity $m 2^{O(\log^3 \log m)}$. 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 [4], [19], [25]. In contrast to these upper bounds, there is a $\Omega(m + p \log p)$ lower bound [14], 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 [4] gave a randomized algorithm that works against an adaptive adversary. The algorithm 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 [15]. 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. [26], 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 [9], [20]. For a detailed overview of research on this problem, we refer the reader to the book by Giorgiou and Shvartsman [17].

Outline. Section II formalizes the model and definitions used in the paper. Section III presents our To-Do Tree algorithm for asynchronous task allocation. Sections IV and V analyze the randomized and derandomized versions of the To-Do Tree algorithm. Full proofs and pseudocode appear in the full version [3].

II. Definitions and Notations

Model. We assume the classic asynchronous shared-memory model [6], [24] with p processes $1, 2, \ldots, p$, where up to $t < p$ processes may fail by crashing. Each process i 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.

III. 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 IV, we prove its correctness in the presence of an adaptive adversary. In Section V, 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. As the execution proceeds, 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, it leaves the min-register 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, as described in [5], where every value read and written is simply subtracted from the default initial value. (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.)

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.)

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 performs a treewalk. 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 the process returns success.

Otherwise, the process 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

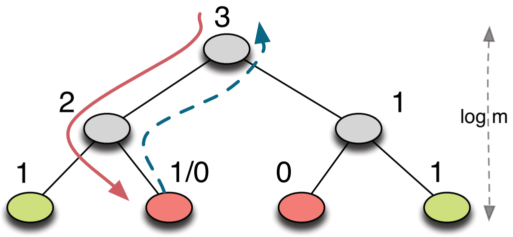

Figure 1. A simple example of a tree-walk operation on a To-Do Tree. The walk descends to reach the red leaf, and then marks up, updating the node min registers. The second red leaf has been counted at the root, as the corresponding min registers have already been updated.

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.

The descent procedure terminates either when the 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 at that leaf, and sets the min register at that leaf to 0, marking the fact that the leaf's tasks have been performed. Then the process begins a *mark-up* procedure from the last node reached during the descent, and repeatedly takes the following steps: (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 when the root has been reached and updated.

IV. Randomized Analysis

It is easy to see that the To-Do Tree ensures that eventually every task is completed. We now analyze its performance, showing that, on termination, the total number of steps taken is $O(m + p \log p \log^2 m)$ and the number of tasks executed is $O(m + p \log p)$. Here, we give an overview of the analysis in case where $m \geq 2p$; see the full version for details. The case where $m < 2p$ follows from a nearly identical analysis.

A. Preliminaries

Fix an arbitrary execution. A leaf is *marked* when a treewalk first reaches it; otherwise it is *unmarked*, or *available*. A leaf is *completed*, or *counted at the root*, when the knowledge that it has been marked has been propagated to the root by some treewalk. The number of *remaining* leaves is the value most recently read or written from the min-register at the root by some complete read or write-min operation.

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

B. Analysis

Treewalk analysis. We say that a 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, with high probability. (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.) Let B_i be the set of complete treewalks in phase i. We need to bound the set of leaves counted by treewalks in B_i.

Intuitively, we would like to think of this procedure as randomly and independently throwing $\Theta(p)$ balls into $V \geq 2p$ bins. In reality, this intuition can be misleading, since the treewalks are *not* independent, nor are they strictly positively or negatively correlated. Moreover, each treewalk may be affected by other treewalks in B_i, or by treewalks that began in a previous phase (and hence are not included in B_i). The key challenge in the analysis resides in accurately bounding the amount of interference between treewalks, and getting right the technical aspects related to independence.

More precisely, we focus on the probability that all treewalks are concentrated among few leaves, and show that this probability is small. We prove the following conditional claim:

Lemma 2: For a fixed set of leaves V not counted at the root prior to phase i, for every treewalk $b \in B_i$: the probability that: (i) b counts a leaf in V, or (ii) that some treewalk that completed its descent prior to b completing its descent counts a leaf in V, assuming (iii) an arbitrary set of concurrent updates to the min-registers, is $\geq |V|/|U_i|$, where U_i is the set of leaves not counted at the root prior to phase i.

The proof is based on the following intuition. (Please see the full version of the paper [3] for a formal proof.) Given a complete treewalk, either the min-registers read by the operation are not changed by concurrent treewalks, in which case the probability distribution over unmarked leaves is roughly uniform, or the min-registers are updated concurrently, in which case the distribution over leaves may be biased arbitrarily. In this second case, the counters are either changed by treewalk operations marking leaves in the set V, in which case the probability of marking a leaf in V only decreases, or by treewalk operations marking leaves outside V, in which case the original walk can only be biased *towards* the set V (since the complement of V shrinks). The claim follows by taking these cases into account. Considering all treewalks in B_i, we conclude:

Lemma 3: 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 $\leq (1 - |V|/|U_i|)^{|B_i|}$.

Phase analysis. At this point, it remains only to show, via a union bound over all possible sets V of appropriate size, that there are $O(p)$ treewalks in each phase:

Lemma 4: For every phase i and for any constant k, $|B_i| = O(p+k)$ with probability at least $1 - (1/2)^{p+k}$.

Notice that, since there are $m/p + \log p$ phases, in some phases (by bad luck) there may be more than p treewalks if p is much smaller than m. We observe that, in expectation, there is $O(1/2^p)$ such wasted work, and by a Chernoff bound, conclude that the total number of treewalks is $O(m+p\log p)$ with probability $1-1/e^{m+p}$. Each treewalk consists of $O(\log^2 m)$ steps, since the To-Do Tree has height $O(\log m)$, and each min-register operation has cost $O(\log m)$ [5]. This yields our main result. When one task is assigned to each leaf, the following holds:

Theorem 5: 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 6: 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 $(1 - 1/e^{m/\log^2 m+p})$.

V. DETERMINISTIC TO-DO TREES

In this section, we derive the existence of a deterministic To-Do Tree algorithm, via the probabilistic method. We begin with an overview of the algorithm, and proceed to divide the analysis into two epochs. After bounding the number of treewalks in each epoch, for a fixed adversarial scheduling, we proceed to take a union bound over all possible schedules, yielding the final result.

A. Flipping coins in advance

Recall that in the randomized To-Do Tree, each process performs repeated treewalks, randomly choosing at each step whether to proceed left or right in its walk down the tree. To derandomize, we assume that all the random choices are made before the execution begins: each process is initialized with a sufficiently long string of "random" bits; it generates $O(\log m)$ random bits for each random number in $[0, 1)$ needed during the protocol. During a treewalk, a process uses these predetermined "random" bits to determine its walk down the tree. Each treewalk uses $\Theta(\log^2 m)$ "random" bits (discarding any unused bits if it terminates its walk before reaching a leaf). Throughout, when we talk about the probability of a certain event, this is in reference to these random choices made before the execution began.

Since the adversary can see all the random bits in advance, it can schedule the processes based on their "future" random choices, attempting to prevent them from making progress. As a result, we can no longer analyze the random choices in the same manner as in Section IV. Even so, we show that with very high probability, the To-Do Tree algorithm still completes all the tasks in $O(m + p\log^5 m\log^2 \max(m, p))$ steps. From this, we conclude (via the probabilistic method) that there exists a good set of input bits for each process, and hence a deterministic To-Do Tree algorithm.

B. 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.

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 in each phase (Lemma 13 and Lemma 14). In fact, the adversary, by his scheduling choices, determines the phase length. 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 (at least $\Theta(p\log^3 m)$) 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 all the treewalks make their random choices in advance, we can analyze the *a priori* distribution of the treewalks, independent of the actual scheduled execution. As such, the treewalks choose tasks uniformly at random from the set of remaining tasks (Claim 8), and the treewalks are "well spread out" in the space of available leaves (Claim 11). 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 10).

We then analyze the behavior of these treewalks in the real tree. We first bound the extent to which the adversary can distort the treewalks (Claim 9). The adversary can "control" a logarithmic factor more leaves than there are treewalks in a phase; we bound the impact of this control, and hence the number of treewalks that can be disrupted (Claim 12). 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.

C. Preliminaries

The execution is divided into two epochs, where the first epoch continues until there are only $O(p \log^3 m)$ leaves left to execute, and the second epoch continues from the end of epoch one until the algorithm completes. Each epoch is divided into phases. In both epochs, each phase is defined such that $\Theta(1/\log m)$ of the remaining work is completed. That is, let s_i be the number of unmarked leaves at the beginning of phase i; the phase ends when $\Theta(s_i/\log m)$ new leaves are counted at the root.

Claim 7: Epoch 1 has $O(\log^2 m)$ phases, and epoch 2 has $O(\log m \log p + \log m \log\log m)$ phases.

We now fix a specific scheduling of the execution. The adversary chooses which processes take steps in which order. We will show that for each schedule, we can bound the number of treewalks in each phase with very high probability. At the end, we will take a union bound over all the (exponentially many) possible schedules. We now give an overview of the analysis; the complete argument can be found in the full version [3].

D. The First Epoch

Fix $\alpha, d \geq 1$ constants. For phase i in epoch one, let $k = \alpha \max(s_i/\log m, p\log^3 m)$. Our goal is to show that k treewalks complete $s_i/(d \log m)$ leaves, i.e., sufficiently many to complete a phase.

For phase i, we define a *reference tree* RT_i in which all the leaves counted at the root when phase i begins are counted, while the remaining leaves are unmarked. Fix a set B_i of treewalks that both start and end in phase i, where $|B_i| = k$, and define the *target* of treewalk 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 8: The probability that a walk $w \in B_i$ targets a specific leaf (in the reference tree) is $1/s_i$.

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 $t_v = j - j/(\beta \log m)$, 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.

We say that a leaf is *in shadow* (in the real tree) if any of its ancestors are muted. We observe that for the duration of phase i, there cannot be too many leaves in shadow:

Claim 9: For $d > 2\beta$ constant, at most $2\beta s_i/d$ leaves are in shadow during phase i of epoch 1.

This follows from noticing that, within a subtree that is muted, there must be one marked leaf for every $\beta \log m$ unmarked nodes, and at most $\Theta(s_i/\log m + p)$ leaves may be marked before the phase ends.

Similarly, we say 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; hence, we make no assumptions on which leaves are reached by walks entering a muted subtree. A subtree contained in a muted subtree is also muted.

Live tree walks. Treewalks that enter muted sections of the tree (i.e., those that target leaves in shadow) may be significantly diverted from their original target. On the other hand, a treewalk that has a non-muted target has a good chance of arriving at its target. (The adversary, however, can influence which treewalks are muted and which are not.) We now analyze how many of the non-muted treewalks will arrive at their target.

We first consider what would happen to a treewalk if, hypothetically, it were executed in a copy of the reference tree in which the adversary could decrement any of the min-registers arbitrarily, with the limitation that no min-register is decreased below its muting threshold. We say that a treewalk is *live* if it still reaches its target in this adversarially perturbed reference tree. We now prove that at least a constant fraction of the k walks in B_i remain live.

Claim 10: Let S be the set of live walks during this phase. With probability at least $1 - (1/e)^{k/16}$, $|S| > k/2$.

This follows from the observation that at each step, each treewalk has a $1/\beta \log m$ probability of dying, as that bounds the maximum disruption the adversary can create at each step.

Sparse tree walks. Consider a list L of all the unmarked leaves at the beginning of phase i, ordered from left-to-right. We say that a subset of treewalks is *sparse* if for every pair of treewalks in the set, there are at least $(\log m)/\alpha - 1$ leaves separating the targets of the treewalks in L. We now argue that there exists a set of at least $\Theta(s_i/\log m)$ treewalks in B_i that are both live *and* sparse.

Claim 11: With probability at least $1 - (1/e)^{k/16}$, there exists a set V of $\alpha s_i/(16 \log m)$ sparse live walks.

The key point is to notice that sparsity and liveness are independent: the distribution of a treewalk over targets is unrelated to whether or not it is alive (which depends on how close the random numbers are to the muting threshold at each node). We begin with the $k/2$ treewalks identified by Claim 10. We group the leaves into bins of size $\log m/\alpha$, and notice that each treewalk effectively chooses a bin at random. By straightforward balls-and-bins analysis, we see that sufficiently many treewalks land in their own bins. Sparsifying the resulting set of treewalks yields the result.

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 the walk is *complete*, meaning that it proceeds down the tree and returns to the root in this phase. 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 min-register that it reads exceeds the muting threshold, then, since the treewalk is alive, it proceeds to its target as it would in the adversarially perturbed reference tree.

Shadowed leaves. Recall that the adversary can choose to shadow $2\beta s_i/d$ of the leaves where $d > 2\beta$ is a constant that we can control. Assume that adversary can choose the leaves to shadow arbitrarily. We define a *contiguous segment* g as a maximal set of consecutive leaves from the list L that are shadowed. We partition the shadowed leaves into contiguous segments $G = \{g_1, g_2, \ldots\}$.

Let us examine a segment g. Assume that some of the walks in V target leaves in the segment g. Then they will enter their respective muted subtrees, and may be arbitrarily diverted away from their target. In the best case for the adversary, they all get diverted to the same leaf. Thus, effectively, for all the walks that are diverted to a single segment, we can only assume that one leaf is counted at the root. On the other hand, since the treewalks in V are sparse, the adversary must shadow a large number of leaves to include more than one treewalk in a segment. Specifically, if segment g contains $t > 1$ treewalks, then the segment shadows at least $\log m(t-1)/\alpha$ leaves. From this we conclude:

Claim 12: 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 $\log m(|H| - 1)/\alpha$.

We can now show that each phase in epoch 1 contains only k complete treewalks, with very high probability:

Lemma 13: 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 $(1/e)^{k/16}$ (Claim 11). 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 *useful* and the others are *wasted*. However, Claim 12 states that for each wasted walk w, the segment which w targets has to contain at least $(\log m)/\alpha$ additional (untargeted) leaves. In turn, based on a simple counting argument, Claim 9 implies that the number of wasted walks in V is at most $O(\frac{s_i}{\log m})$. Therefore, the number of walks in V that are either not targeting leaves in shadow or are useful is at least $\left(\frac{\alpha s_i}{16 \log m}\right) - \left(\frac{2\alpha\beta s_i}{d \log m}\right) \geq \frac{s_i}{d \log m}$, where we have chosen α and d appropriately. This occurs with probability at least $1 - (1/e)^{k/16}$. Since no two such walks may hit the same leaf, for sufficiently large α, the main claim follows. ■

E. The Second Epoch

The analysis of the second epoch is similar to the first, except we can no longer assume that treewalks are sparse: there are no longer enough available leaves. Also, to ensure exponentially high probability in p, we need a sufficient number of treewalks per phase. The second epoch starts with at most $O(p \log^3 m)$ leaves uncounted at the root. We show that each phase in this epoch contains $O(p \log^3 m)$ complete treewalks.

Lemma 14: 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 $(1/e)^{(\alpha - c)p \log^3 m}$.

F. 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. We now count the number of possible schedules. We give an overview of the argument for $m \geq p$; the other case is similar, and can be found in the full version.

Since each treewalk uses a fixed number of random bits, we need only enumerate the number of different ways in which treewalks may be assigned to phases. We assume the adversary chooses how many walks to schedule for each process and their interleaving among processes. (For each process, the adversary can only schedule walks in order.)

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. These treewalks can be allocated to any subset of the processes in any quantity. For a constant $c' > 1$, the number of possible combinations is upper bounded by $\binom{\alpha m/\log m + 2p}{2p} \leq 2^{c' p \log m}$.

From Claim 7, we know that there are $O(\log^2 m)$ phases in total. The total number of interleavings is bounded by the product of the interleavings in each phase, and hence we bound the number of schedules by:

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

with $c \geq 1$ constant.

By Lemma 13 and Lemma 14, there exists a constant $\alpha \geq 1$ such that, for a given scheduling of treewalks, the probability that any of the $O(\log^2 m)$ phases fails to complete with the specified number of treewalks is at most $O(\log^2 m)\,(1/2)^{\alpha p \log^3 m} \leq (1/2)^{(\alpha-1)p \log^3 m}$. By a union bound, the probability that there exists exists an interleaving of treewalks for which some phase fails is $\leq (1/2)^{(\alpha-c-1)p\log^3 m} < 1$, for $\alpha > c+1$. Thus, there exists a sequence of random bits for each process for which all phases are *successful*.

The last step is to upper bound the total work performed during an execution in which all phases are successful. The total number of treewalks (and hence tasks executed) is at most: $O((m - p\log^4 p) + p\log^3 m(\log m \log p + \log m \log\log m))$. Each treewalk has cost $O(\log^2 m)$.

By assigning $\log^2 m$ tasks to each leaf in the tree, we obtain that the total amount of work is $O(m/\log^2 m + p\log^4 m(\log p + \log\log m)\log^2 m) = O(m + p\log^6 m(\log p + \log\log m))$. This yields our main theorem:

Theorem 15: For general m and p, there exists a deterministic To-Do Tree algorithm with total work and number of tasks executed $O(m + p\log^5 m(\log p + \log\log m)\log\max(m,p))$.

VI. Conclusion

We presented 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 [14] 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, the at-most-once problem [22] and *do-most* [20], in which only a fraction of the tasks need be performed.

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] D. Alistarh, M. A. Bender, S. Gilbert, and R. Guerraoui. How to allocate tasks asynchronously. Technical report, EPFL, 2012. http://infoscience.epfl.ch/record/180499.

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

[5] J. Aspnes, H. Attiya, and K. Censor. Polylogarithmic concurrent data structures from monotone circuits. *Journal of the ACM*, 59(1):2:1–2:24, Feb. 2012.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

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

[27] 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.

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