Dynamic Task Allocation in Asynchronous Shared Memory

Dan Alistarh* MIT

James Aspnes† Yale

Michael A. Bender‡ Stony Brook University & Tokutek

Rati Gelashvili§ MIT

Seth Gilbert¶ NUS

Abstract

Task allocation is a classic distributed problem in which a set of p potentially faulty processes must cooperate to perform a set of tasks. This paper considers a new *dynamic* version of the problem, in which tasks are injected adversarially during an asynchronous execution. We give the first asynchronous shared-memory algorithm for dynamic task allocation, and we prove that our solution is optimal within logarithmic factors. The main algorithmic idea is a randomized concurrent data structure called a *dynamic to-do tree*, which allows processes to pick new tasks to perform at random from the set of available tasks, and to insert tasks at random empty locations in the data structure. Our analysis shows that these properties avoid duplicating work unnecessarily. On the other hand, since the adversary controls the input as well the scheduling, it can induce executions where lots of processes contend for a few available tasks, which is inefficient. However, we prove that *every* algorithm has the same problem: given an arbitrary input, if OPT is the worst-case complexity of the optimal algorithm on that input, then the expected work complexity of our algorithm on the same input is $O(OPT \log^3 m)$, where m is an upper bound on the number of tasks that are present in the system at any given time.

1 Introduction

Sharing work efficiently and robustly among a set of agents is a fundamental problem in distributed computing. The problem is all the more challenging when there is *heterogeneity*, either on the workers' side, since individual agents may have varying speed and robustness, or because of uneven workloads. In large-scale systems, heterogeneity is the norm. A further challenge for task allocation is the fact that scheduling must often be decentralized: the designer cannot afford a centralized scheduler either because communication costs would be too high, or because of fault-tolerance concerns.

Considerable research, e.g. [4,5,11–13,16,19,22–24], has focused on algorithms and lower bounds for the *asynchronous* version of task allocation, also known as *do-all* [17], or *write-all* [19], where processes move at arbitrary speeds and are crash-prone. Task allocation is closely connected to many other fundamental distributed problems, such as mutual exclusion [10], distributed clocks [8], and shared-memory collect [3]. The book by Georgiou and Shvartsman [17] gives a detailed history of the problem.

Most of the theoretical research on task allocation has looked at the *one-shot* version, where m tasks are available initially, and the computation ends when all tasks are performed. (Notable exceptions are references [16] and [15], which consider task injection under strong timing assumptions.)

This paper formalizes *dynamic task allocation*, which captures the (parallel) producer-consumer paradigm. We are given p asynchronous processes that cooperate to perform tasks, while new tasks are inserted dynamically by the adversary during the execution. This problem has attracted significant interest among practitioners do to the importance of the producer-consumer problem. (See the end of this section for a discussion of such work.)

A dynamic task allocation object supports two operations: DoTask, and InsertTask. The input to an execution is a sequential list of operations, where each DoTask must return the index of the task performed (or empty in case there are no more tasks left), and each InsertTask returns success once the task has been inserted. We require both operations to be linearizable.

*This author was supported by the SNF Postdoctoral Fellows Program, NSF grant CCF-1217921, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

†Supported in part by NSF grant CCF-0916389.

‡This research was supported in part by NSF grants CCF 1114809, CCF 1217708, IIS 1247726, and IIS 1251137.

§This work was supported in part by NSF grants CCF-1217921, CCF-1301926, DoE ASCR grant ER26116/DE-SC0008923, and by grants from the Oracle and Intel corporations.

¶Supported by Singapore AcRF-2 MOE2011-T2-2-042.

(See Section 2 for the precise semantics of these operations.) We assume that m is an upper bound on the number of tasks present in the data structure at any given time. We focus on a natural extension of the *work* performance metric [17], which counts the total number of shared-memory steps.

The problem is especially challenging in an asynchronous setting since the adversary sees the processes' coin flips, controls the scheduling, and the process crashes, *and* chooses the input. In fact, the problem may appear inapproachable: for instance, the adversary can easily build an input and a schedule such that, at regular time intervals, there is only *one* task available in the system, and p processes are competing to perform tasks. Since any solution must be fault-tolerant, each process must try to perform the one available task. Thus, we always spend at least p work to perform only one task! A natural question is: can anything efficient be done under such harsh conditions? In general, how well can an algorithm do for an arbitrary input I?

Contribution. In this paper, we show that efficient solutions do exist, by presenting an algorithm which is optimal within logarithmic factors for every input I. More precisely, assuming an optimal algorithm OPT for the problem, if w is the amount of work that OPT needs to perform on input I in a worst-case schedule, then, on the same input I, our algorithm will perform expected $O(w \log^3 m)$ total work, and $O(w \log^3 m \log p)$ work with high probability. To the best of our knowledge, this is the first algorithm which solves dynamic task allocation in an asynchronous system.

Our algorithm is based on a data structure called a *dynamic to-do-tree*. The underlying intuition is that, to minimize contention, processes should pick tasks to perform *uniformly at random*, and attempt to insert new tasks at uniform random available locations in the data structure. We achieve this by fixing a set of m locations (each of which will be associated to a task), and build a binary tree on top of the locations. Each node in the tree has two counters: one for the number of successful insert operations performed in its subtree, and one for the number of successful remove operations. Intuitively, the difference between the insert count and the remove count, called the *surplus* at the node, gives the number of available tasks in the subtree.

When calling DoTask, a process starts at the root and walks down the tree looking for an available task. It decides whether to go left or right by checking the surplus at the left and right children. More precisely, if s is the left surplus and s' is the right surplus, then the process goes left with probability $s/(s+s')$, and right otherwise. (If $s+s'=0$, the process re-traces its steps to the root and tries again.) Once a task is executed at a leaf, the process walks back up to the root, updating the counters on the way. The InsertTask operation is completely symmetric, except that it follows the *space* at each node, which is the *complement* of the surplus at a node. Both operations are linearizable and lock-free. The algorithm ensures that each inserted task is eventually performed with probability approaching 1.

Structurally, the dynamic to-do tree is relatively simple, and borrows ideas from other shared-memory constructions such as the poly-logarithmic snapshots of Aspnes et al. [6] and the one-shot task allocation algorithm of Alistarh et al. [4]. *Progress trees* are a natural idea for task allocation and variants have been analyzed previously, e.g. [4, 11, 24], however this is the first instance of a *dynamic* progress tree, which supports concurrent insert and remove operations. Instead of using a single progress tree (sufficient for one-shot algorithms) we combine two *dual* progress trees: one for tracking insertions, and one for tracking removals. The two implementations are "glued together," and interact in non-trivial ways in the actual executions. The interactions between inserted tasks and removed tasks add significant complications.

Our main technical contribution is in showing that this simple algorithm works, and is in fact optimal within logarithmic factors. Even though the intuition behind the data structure is clean, the analysis of concurrent executions can be quite complex. For instance, the idea that process q picks a task to perform "at random," minimizing contention, is appealing conceptually. If the operations were performed sequentially, or if the processes took steps synchronously, it would be true and the analysis would be straightforward.

Unfortunately, due to asynchrony and dynamic operations, this intuition of random choices is not accurate. When one operation is executing, the $p-1$ other processes are removing and inserting tasks at the same time, changing the counters that the process reads. In fact, the adversary may easily adapt the schedule so that a certain process is *prevented* from performing a certain task with good probability. (For example, whenever a process is about to reach a task to perform, it is delayed until some other process executes that task.) Thus, most of the technical effort in the paper is spent in building a framework to analyze the processes' random choices under asynchronous scheduling.

Instead of proving that individual processes make progress (a losing proposition), we focus on *blocks* of tree walks of carefully chosen size. We prove that their *collective behavior* is random, in that, with high probability, the adversary cannot confine them to reaching a small subset of the available tasks. The block size is a trade-off between the probability that a large subset is covered, and the amount of wasted work. Interestingly, this argument holds both for insert and remove walks, though the analysis details diverge depending on the operation type.

Another technical challenge is that known concurrent counter constructions are not well suited for large numbers of concurrent insertion and removal operations. The poly-logarithmic MaxArray of [6] only supports a *bounded* number of operations. Also, it allows us to read the two sub-counters atomically, but not to *update* both atomically: this leads to an inherent asymmetry between the two types of operations, where inserts may propagate up the tree faster than the respective removes. We resolve the former issue by building an unbounded-use MaxArray using atomic compare-and-swap operations. The latter is solved by an analytic technique which accounts for the distribution skews caused by using weak counter objects.

The analysis suggests that the key parameter is the ratio of available processes to available tasks throughout an execution. Specifically, if q processes are performing DoTask operations and there are only $s \leq q$ tasks available at some point, then the algorithm spends $\Theta(q/s \log^3 m)$ work in expectation to perform $\Theta(s)$ tasks.

Our competitive analysis shows that part of this cost is inherent. Fixing an input I, we run both our algorithm and the hypothetical optimal algorithm OPT in parallel. We carefully build a schedule for OPT such that, if our algorithm reaches a situation where q processes must perform $s \ll q$ tasks, the optimal algorithm ends up in a similar scenario. Both algorithms will then waste similar amounts of work to perform the tasks. The difference between the two algorithms will be the cost of tree walks–$O(\log^3 m)$ each–which our algorithm pays and the optimal algorithm may not. We obtain a lower bound on the cost of OPT on I, in terms of a worst-case execution for our algorithm on I. The notion of competitiveness with a fixed input is new for concurrent algorithms: previous work, e.g. [3], defined competitiveness with respect to a fixed adversarial schedule.

In sum, we show that processes can cooperate to perform work efficiently even in strongly adversarial conditions. Our algorithm ensures global progress, and employs the common atomic read, write, and compare-and-swap (CAS) operations. Note that we assume that the compare-and-swap operation is available in hardware. This holds for virtually all modern architectures, however this assumption makes our computational model strictly stronger than the asynchronous read-write model [18] used by previous work on write-all, e.g. [4, 11, 24]. The complexity bounds are amortized, and we guarantee that each inserted task is eventually performed by some process.

Previous work. Task allocation is also known as *do-all* [17]; the shared memory variant is also called *write-all* [5, 19]. The recent book by Georgiou and Shvartsman [17] gives a detailed overview of work on the problem. In brief, the shared-memory version was introduced by Kanellakis and Shvartsman [19] in the context of PRAM computation: in this instance, each task is a register, which must be flipped from 0 to 1. There has been significant follow-up work on the topic, e.g. [5, 11, 12, 22–24]. However, most of this work has focused on the *one-shot* version of the problem, in which the set of tasks is fixed initially, and computation ends when all tasks have been performed. Alistarh et al. [4] used a variant of the progress tree to obtain the most efficient task allocation algorithm to date: the randomized version of the algorithm ensures expected total work $O(m + p \log p \log^2 m)$ for m tasks using p processes, while a (non-explicit) deterministic version ensures total work $O(m + p \log^7 m)$ for $m \geq p$. This improves on previous algorithms by Kowalski and Shvartsman [22], Malewicz [23], and Chlebus and Kowalski [12].

Our algorithm has expected cost $O(m + p \log p \log^3 m)$ in an execution where only insertions or removals are performed. The best known (one-shot) lower bound for the problem is of expected $\Omega(m + p \log m)$ shared-memory steps [24]. Recent work on the do-all problem considered *at-most-once* semantics in shared memory, e.g. [20, 21], instead of at-least-once. Assuming stronger synchronization primitives than [20, 21], our solution gives exactly-once semantics (see Section 2 for details).

A variant of *dynamic* do-all, where tasks can be injected dynamically during the execution, has been considered recently by Georgiou and Kowalski [15] in the message-passing model with process crashes and restarts, assuming a synchronous round structure. The authors identify trade-offs between efficiency and fault-tolerance, and introduce a framework for competitive analysis where the efficiency of an algorithm is measured in terms of the number of pending tasks at the beginning of a computation round. Georgiou et al. [16] had previously considered the iterated version of the problem under process crashes. In this paper, we assume a strictly harder adversarial model, since computation is completely asynchronous, and the adversary is adaptive. On the other hand, our algorithm only provides probabilistic guarantees.

A more applied research thread has looked at efficient shared-memory data structures with *set* semantics, also known as *(task) pools*, e.g. [1, 2, 9, 25], scalable non-zero indicators (SNZI) [14], or combining funnels [26]. In general, these references emphasize the practical performance of the data structure, and do not provide complexity upper bounds. One exception is the CAFE data structure [9], where removes take $O(\log^2 p)$ steps with high probability, and inserts eventually terminate—on the other hand, their amortized complexity may be linear.

Roadmap. Section 2 describes the model and the problem statement. The algorithm is described in Section 3, while Section 4 gives the analysis of the algorithm. We prove that the algorithm is optimal within logarithmic factors in Section 5. The unbounded MaxArray construction is given in Appendix Section A.

2 System Model and Problem Statement

Model. We work in the standard asynchronous shared memory model. We have p processes which communicate through registers, on which they can perform read, write, and compare-and-swap (CAS) operations. Each process is assumed to have a unique identifier from an unbounded namespace. Each process has at its disposal a (local) random number generator. Specifically, the call random(a, b) returns a random integer chosen uniformly from the interval $[a, b]$. At most $p-1$ processes may fail by crashing. A crashed process stops taking steps for the remainder of the execution.

The scheduling of process steps and their crashes are controlled by a *strong adaptive adversary*. Specifically, the adversary can examine the state of the processes (including random coin flips) at any point in time, and decide on the schedule accordingly.

Dynamic Task Allocation. In this problem, the p processes must perform a set of tasks which are dynamically injected by the adversary over time. For simplicity, we assume that there is a bound $m \geq p$ on the number of tasks that may be available at any point in the execution, and that each task has a unique identifier $\ell \geq 0$. (As suggested by an anonymous reviewer, a simple unique task identifier scheme can be implemented by assigning to each task an identifier of the form $(id, count)$, where id is the id of the process to which the task is assigned, and $count$ is the value of a local per-process counter, which is incremented on each newly inserted task.)

A process may perform two types of operations. The DoTask operation performs a new task and returns the index of that task, while the InsertTask(ℓ) operation inserts a task ℓ to be performed. The input is a string of DoTask and InsertTask(ℓ) operations to be executed. When a process completes its current operation, it is assigned the next operation in the string.

We assume that the input at each process and the scheduling are controlled by the adversary. The input ensures the following properties:

- (Task Upper Bound) There can be at most m tasks available at the same time. More precisely, we require that, for every contiguous subsequence of the input, the number of InsertTask operations minus the number of DoTask operations in the subsequence is at most $m - 2p$. (See Appendix Section B for a discussion of this limitation.)
- (Well-formedness) No two InsertTask calls get the same task identifier as argument.

For clarity, we fix an interface by which threads may perform a task, or insert a new task. We assume that each task ℓ to be performed is associated to a memory location M. (Over time, a memory location can be associated with multiple tasks.) The task ℓ can be performed atomically by a process by calling a special TryTask operation on the memory location M associated to the task.

Out of several processes calling TryTask(M) concurrently, only one receives success and the index ℓ of the task, whereas all the others receive a failure notification. This ensures that only a single thread may actually perform the task. A process returns the task index ℓ from DoTask if and only if it has received success from the TryTask(M) call.

Note that if such a *compare-and-perform* operation is not available, then we can use a compare-and-swap to *assign* the task to the winning process. This changes the semantics of the problem from *exactly-once* to *at-most-once* [21]: the algorithm can only guarantee that all but $p-1$ tasks are performed, since the adversary can stop a process between the point it has been assigned to a task, and the point when it performs it.

A task can be associated with a memory location M by a special PutTask$(M, task)$ operation. The operation returns success if the task has been associated with the location, and failure if the location was already associated with another task.

To illustrate, consider the classic WriteAll problem [19], where threads must change the values of a set of memory locations from 0 to 1; in this case, the TryTask(ℓ) operation attempts to perform a CAS from 0 to 1 on the location ℓ, whereas the PutTask$(M, task)$ attempts to perform a CAS from 1 to 0 on the location. The algorithmic challenge is to distribute the calls in such a way as to minimize the amount of wasted work.

We require the DoTask and InsertTask operations to be linearizable. More specifically, for every execution of the data structure, there exists a total order on the completed DoTask and InsertTask operations. A task is *done* if its index has been returned by a process after a DoTask call. A task is *available* if it has been inserted, but not done. The total order must verify the following requirements: every done task has been inserted (validity); every available task is eventually done (fairness); each task is performed exactly once (uniqueness).

Complexity Measures. The complexity measure we consider is *work*, or *total step complexity*, that is, the total number of shared-memory operations (read, write, or compare-and-swap) that processes take during an execution. Since our algorithms are randomized, total work is a random variable in the probability space given by the processes' coin flips.

Auxiliary Objects. An object r of type $\mathsf{MaxArray}_{K \times H}$ [6] supports three linearizable operations: $\mathsf{MaxUpdate0}(r, v)$, where the value v is between 0 and $K-1$, $\mathsf{MaxUpdate1}(r, v)$, where the value v is between 0 and $H-1$, and $\mathsf{MaxScan}$ with the following properties: (i) $\mathsf{MaxUpdate0}(r, v)$ sets the value of the first component of r to v, assuming $v < H$; (ii) $\mathsf{MaxUpdate1}(r, v)$ sets the value of the second component of r to v, assuming $v < K$; $\mathsf{MaxScan}(r)$ returns the value of r, i.e. a pair (v, v') such that v and v' are the largest values in any $\mathsf{MaxUpdate0}(r, v)$ and $\mathsf{MaxUpdate1}(r, v')$ operations that are linearized before it.

The results of two $\mathsf{MaxScan}$ operations are always comparable under the standard $\leq$ partial order. Note that the implementation of MaxArrays given in [6] is *limited-use*, as it limits the maximum number of update operations that can be applied during an execution. The step complexity of MaxArrays is poly-logarithmic in H and K. In particular, the $\mathsf{MaxUpdate0}$ operation has cost $O(\log H)$, the $\mathsf{MaxUpdate1}$ operation has cost $O(\log K)$, and the $\mathsf{MaxScan}$ operation has cost $O(\log H \log K)$ [6]. We give an *unbounded* MaxArray construction with polylogarithmic amortized complexity in Appendix A.

3 The Dynamic To-Do Tree

The main data structure used for keeping track of tasks is a binary tree with m leaves, where each leaf is either empty or associated with a task that is available to perform. Each tree node contains a two-location *unbounded* $\mathsf{MaxArray}$. (The specification of a $\mathsf{MaxArray}$ is given in Section 2, and the implementation is described in Appendix A.) The first component of the MaxArray, called the *insert count*, tracks the number of tasks successfully inserted in the subtree. The second component is the *remove count*, and tracks the number of successful DoTask operations in the subtree.

For any node v in the tree, the first component of the associated $\mathsf{MaxArray}$ counts the total number of successful $\mathsf{InsertTask}$ operations performed in the subtree rooted at v. The second component of the $\mathsf{MaxArray}$ counts the total number of successful DoTask operations performed in the subtree rooted at v. In addition to the $\mathsf{MaxArray}$, each leaf also contains an array that stores available (and completed) tasks.

We fix $m = p^\beta$ with $\beta > 1$ constant. Using the construction in Appendix A, the (amortized) complexity of $\mathsf{MaxUpdate0}$ and $\mathsf{MaxUpdate1}$ operations is $O(\log m)$, and the complexity of $\mathsf{MaxScan}$ is $O(\log^2 m)$.

Intuitively, if the MaxArray at vertex v returns the pair (x, y), then there are $(x - y)$ tasks available in the sub-tree rooted at v, since there have been x tasks inserted and y tasks performed. (Formally, we must account for concurrent operations.) We call this difference $(x - y)$ the *surplus* at vertex v, and denote it by u_v. Symmetrically, if node v has height h, we call the complement $(2^h - u_v)$ the *space* at node v. This is an estimate for the number of tasks that can still be inserted in the subtree rooted at that node.

For simplicity, we assume that the tree is initially full of tasks, i.e., each leaf has a distinct task, each internal node at height h has surplus 2^h, and each node has zero space. (However, our analysis works from any initial configuration.)

The pseudocode for the DoTask procedure is given in Figure 1. In brief, a process performs a task as follows: it first checks the surplus at the root. If this is zero, then there are no tasks to perform, and the process returns. Otherwise, the process proceeds down toward a leaf. At each node, it reads the surplus at the right child into x (using a single $\mathsf{MaxScan}$ operation), and the surplus at the left child into y. It then proceeds left with probability $x/(x+y)$, and right otherwise. If $x + y = 0$, then the process backtracks towards the root.

Upon reaching a leaf, the process reads the MaxArray at the leaf into (x, y). The value y indicates the number of tasks that have been performed at this leaf, so the process attempts to perform task in slot $y+1$ of the array by executing $\mathsf{TryTask}$. Irrespective of wehther it successfully acquired a task to perform through $\mathsf{TryTask}$, the process then walks back up to the root, updating the counts at each $\mathsf{MaxArray}$. If it succeeded in performing a task, it returns. Otherwise, it proceeds to perform another treewalk.

Inserting a task is symmetric, where the choice of which child to visit during the treewalk is based on the space rather than the surplus. On reaching a leaf, a process calls $\mathsf{PutTask}$ to add the new task to the leaf's task array in position $x+1$, where (x, y) is the last pair scanned from the MaxArray. The pseudocode for the

```
1  procedure DoTask⟨ ⟩
2    while true do
3      v ← root
4      if v.surplus() ≤ 0 then return ⊥
       /* Descent */
5      while v is not a leaf do
6        (x_L, y_L) ← MaxScan(v.left)
7        (x_R, y_R) ← MaxScan(v.right)
8        s_L ← min(x_L − y_L, 2^height(v))
9        s_R ← min(x_R − y_R, 2^height(v))
10       r ← random(0, 1)
11       if (s_L + s_R) = 0 then Mark-up(v)
12       else if (r < s_L/(s_L + s_R)) then
13         v ← v.left
14       else v ← v.right
       /* v is a leaf */
15     (op, y) ← MaxScan(v)
16     (flag, ℓ) ← TryTask(v.tasks[op])
       /* Update removal count */
17     MaxUpdate1(v.MaxArray, op)
18     v ← v.parent
19     Mark-up(v)
20     if flag = success then return ℓ
```

```
1  procedure InsertTask⟨task⟩
2    while true do
3      v ← root
       /* Descent */
4      while v is not a leaf do
5        (x_L, y_L) ← MaxScan(v.left)
6        (x_R, y_R) ← MaxScan(v.right)
7        h = height(v)
8        s_L ← 2^h − min(x_L − y_L, 2^h)
9        s_R ← 2^h − min(x_R − y_R, 2^h)
10       r ← random(0, 1)
11       if (s_L + s_R) = 0 then Mark-up(v)
12       else if (r < s_L/(s_L + s_R)) then
13         v ← v.left
14       else v ← v.right
       /* v is a leaf */
15     (x, op) ← MaxScan(v)
16     flag ← PutTask(v.tasks[op + 1], task)
       /* Update insertion count */
17     MaxUpdate0(v.MaxArray, op + 1)
18     v ← v.parent
19     Mark-up(v)
20     if flag = success then return success
```

```
1  procedure Mark-up(v)
2    if v is not null then
3      (x_L, y_L) ← MaxScan(v.left)
4      (x_R, y_R) ← MaxScan(v.right)
5      MaxUpdate0(v, x_L + x_R)
6      MaxUpdate1(v, y_L + y_R)
7      Mark-up⟨v.parent⟩
```

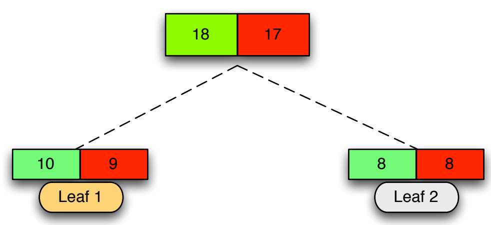

Figure 1: Pseudocode for DoTask, InsertTask, and Mark-up. In the dynamic to-do tree, each node maintains an insert count and a remove count. In the example, leaf 1 has surplus 1 and space 0, while leaf 2 has surplus 0 and space 1. Processes update the tree so that the counts at a node reflect the counts at descendants.

InsertTask procedure is given in Figure 1.

Parameter values. Our algorithm has parameters m, the maximum number of tasks in the data structure, p, the number of processes, and $H = K$, the number of operations after which the unbounded MaxArray is "refreshed." In the following, we assume that $m = p^\beta$, for $\beta > 1$ constant, and that $H = K = p^\alpha$, for $\alpha > \beta > 1$. For values of α less than β, the cost of the unbounded MaxArray may dominate the cost of the tree traversals. If $m < p$, the cost of the MaxArray operations is poly-logarithmic in p.

4 Analysis

In this section, we analyze the correctness and performance properties of the algorithm. We begin by defining some auxiliary notions that will be useful in the rest of the proof.

4.1 Preliminaries Recall that we consider two types of operations: DoTask and InsertTask. Each operation is composed of *treewalks* that begin at the root, walk down to a leaf (or some intermediate node), and return to the root. A DoTask treewalk is *successful* if it succeeds in its TryTask operation or if it sees 0 surplus at the root in line 4. An InsertTask treewalk is *successful* if it succeeds in its PutTask operation. Each DoTask or InsertTask operation by a correct process ends with a successful treewalk. Notice that each successful operation can be associated with a unique array slot number at the leaf corresponding to the task it inserted or removed.

Operation Propagation. The counter values at the nodes are continually updated during an execution. We associate the node counts with tasks whose insertion or completion has been propagated up to the node, as follows.

First, notice that the *insert count* (i.e., the value of the first max register) and the *remove count* (i.e., the value of the second max register) are always monotonically increasing, by the properties of a max array. We define the event that a task has been *counted at a node* recursively, as follows. A task is counted at a leaf ℓ once some process completes a MaxUpdate operation on the leaf max array component corresponding to the task type (the insert count, or the remove count) with a value that is at least the index of the operation in the *tasks* vector.

Next, we define counting insert tasks at an arbitrary internal node z. (Counting remove tasks is symmetric.) We group the MaxUpdate operations on the first component of z according to the value they write: let O_v be the set of operations writing value v to the first component of the max array at z. We sort the operations in O_v by their linearization order. Let op_v be the first operation in O_v to be linearized. Operation op_v is the only in O_v which may count new tasks at node z; we will not count any new operations at this node after any other operation in O_v. Intuitively, the insert count must increase if a new operation is counted.

Newly counted tasks are assigned to operation op_v as follows. First, if op_v is preceded in the linearization order by some other operation op_u writing a value $u > v$, then op_v does not count any new task at z.

Consider now the set of operations which update the value of either the left or right child of the current node. Importantly, notice that these operations can be ordered by the values they read from the left and right children when updating the value of the node. (This is a consequence of the fact that they first update the child, then read the left child, then read the right child.) In brief, each such operation "sees" the updates performed by previous operations. We can therefore enumerate these operations in increasing order of the values they write to the node z, as $o_1, o_2, \ldots, o_k$. If op_v is such an operation, recall that it writes $v = x + y$ to z, where x is the insert count it read at the left child of z, and y is the insert count it read at the right child of z. Let L be the set of insert tasks counted at the left child of z when the node has insert count x, and let R be the set of insert tasks counted at the right child of z when the node has insert count y. Also, let C_z be the set of tasks counted at z *before* op_v is linearized. Then the set of tasks counted at z after op_v is linearized is $C_z \cup R \cup L$. In other words, the tasks in $(R \cup L) \setminus C_z$ are the new tasks counted at z. Notice that $(R \cup L) \setminus C_z \neq \emptyset$, since op_v is the first operation to write a value $\geq v - 1$ to the insert count of z.

On the other hand, notice that there also exist operations q which update the value at z, but do not update the values at the left and right children of z. Notice that each such operation can always be placed between two updating operations o_j and o_{j+1}, as defined above, depending on the values it reads from the registers at the left and right children. If the operation q sees the update of o_j but not the one by o_{j+1}, then its associated set of insert tasks is the same as that o_j. However, there exist interleavings where the operation sees the update of o_{j+1}, but not that of o_j. For example, this occurs if o_j increases the insert count at the left child, and o_{j+1} increases the insert count at the right child. By the structure of the algorithm, we always have that either o_{j+1} sees o_j's update, or vice-versa, however q might see o_{j+1}'s update in the right child, but its read of the left child can precede o_j's update. In this case, we need to assign tasks to q such that its update does not break the property that each task is counted at a node by the time its max array update completes at the node. First, notice that, for any such operation q, we can identify the updating operations o_j and o_{j+1} based on the values q reads from the left and right children. Let $v = x+y$ be the value that q writes to z, let $x' > x$ be the value that o_j writes to the left child, and let $y' < y$ be the value that o_j reads from the right child. (The converse case is symmetric.) Let L be the set of insert tasks counted at the left child of z when the node has insert count x', let R be the set of insert tasks counted at the right child of z when the node has insert count y, and let $R' \subseteq R$ be the set of insert tasks counted at the right child of z when the node has insert count $y' < y$. Also, let C_z be the set of tasks counted at z *before* q's update is linearized. Then we allocate to q the set of tasks in L, plus a set S of $(x + y) - (x' + y')$ tasks from $R \setminus R$, taken in the order in which they have been counted at the right child, and then by task identifier. Then the set of tasks counted at z after q's update is $R \cup S \cup C_z$. These cases cover all possible types of update operations.

We say that a task is *counted at a node* as soon as the MaxUpdate operation counting the task is linearized. In particular, the updates counting task k at some node are not necessarily performed by the operation performing the task. The counting scheme has the following properties.

Lemma 1. *Let v be a tree node at height h. Consider a* MaxScan *operation ϕ at v.*

- *If ϕ returns (x, y), then there exists a set I_v of x successful* InsertTask *operations, and a set D_v of y successful* DoTask *operations that have been counted at v by the end of the* MaxScan *operation. If ϕ is linearized after z* InsertTask *(or* DoTask*, respectively) operations have been counted at v, then it returns a value $\geq z$ in the corresponding* *entry of the output tuple.*

- *If the remove count at a node v is x, then the insert count at that node is also at least x. The surplus at a node is greater than* 0*.*
- *Let w be a walk, let T_w be the set of tasks associated with the walk at node z, and let z' be the parent of z. Then all tasks in T_w are counted at z' by the time w's* MarkUp *operation completes at z'.*

Proof. First, notice that, by the structure of the counting procedure, no task is counted twice at a node. (This can be shown formally by induction over the tree height.)

For the first claim, consider the returned insert count x. There must exist a MaxUpdate0 operation which wrote x to the max array, and no previously linearized operation wrote a larger count. Therefore there exists a *first* such operation op_x writing the value x. By definition, x distinct tasks are counted at x after this operation is linearized, as required. The argument for remove counts is symmetric. The second statement follows by the linearizability of the max array at v.

We prove the second claim by induction on the height of v. If v is a leaf, assume for contradiction that the insert count at v is $x-1$, whereas the remove count is x. Therefore, there must exist a DoTask operation which wrote remove count x. But this implies that this operation performed a TryTask call on $v.tasks[x]$, which implies that it read insert count x. This is impossible, since the count in each max register component is monotonically increasing, by definition.

If v is an internal node, assume for contradiction that there exists a time t when some node has remove count larger than the insert count, and let v be that node. First, note that this may only occur after an update of the remove count at v, since the insert count is monotonically increasing. Therefore, there exists a MaxUpdate1 operation which writes value $y = y_1 + y_2$ to the remove count, having read values y_1 and y_2 for the remove counts at the two children, respectively. Recall, however, that this operation has first performed a MaxUpdate0 operation on the insert count, writing value $x = x_1 + x_2$. Moreover, by the inductive hypothesis, $x_1 \geq y_1$, and $x_2 \geq y_2$. Therefore, the value of the insert count at time t is at least $x \geq y$, a contradiction. The surplus is positive by the argument above. The third claim follows by the structure of the counting procedure. □

Given the previous definitions and claims, we can linearize every operation at the point when it is counted at the root, taking care to properly order operations linearized by the same MarkUp.

Lemma 2. *For any execution of the algorithm, there exists a total order on the completed operations which verifies the* validity *and* uniqueness *properties.*

Proof. The fact that each task is inserted/removed only once is ensured by the semantics of addition / removal at the leaves. We can prove by induction over the height of the tree that the insert operation for a task is always counted at a node *before* the corresponding remove for that task. At the leaf, this holds since the insert count must first be incremented before the remove operation sees the task as inserted (see line 17 of InsertTask). For an internal node, this holds since every operation updates the insert count *before* the remove count. If a remove operation is counted at the parent through a MaxUpdate1, then the corresponding insert must have been counted at the child, by the induction step. Therefore, the intervening MaxUpdate0 must have been scheduled at the parent, counting the corresponding insert operation at the parent. Applied at the root, this property ensures that the *validity* condition holds.

Finally, we need to ensure that the linearization order ensures that there are never more than m tasks inserted in the data structure. For this, we delay the linearization of an insert operation until after the preceding remove operation at the leaf has been linearized. This is always possible since either the remove completes *before* the insert starts, or the insert and the preceding remove are concurrent. □

4.2 Performance Recall that an input I is a sequence of InsertTask and DoTask operations. It defines the order and type of the operations that the algorithm will perform. We say that a process is *busy* if it is currently executing an operation, otherwise it is *available*. Initially, all the processes are available. At any point during the execution, the adversary can pick an available process and assign it to the next operation in I, or it can choose an already assigned process and let it take a step. Once a busy process returns from an operation, the process becomes available again. A task is *available* if it has been inserted, but not removed. Otherwise, the task is *done*.

Notice that, at any point in the execution, the input I uniquely determines the next operation that will be assigned to some process by the adversary. Since the code is symmetric, it makes no difference to the adversary which available process is assigned the next operation. Therefore, in the following, we shall assume without loss of generality that, once a process becomes available, it is immediately re-assigned to the next unassigned operation in the input string. This implies that each process is always assigned to either an InsertTask or a DoTask operation.

In the following, we fix a constant $\alpha \geq 1$, and an input I. Let the algorithm run against the worst-case adversary on I and denote the resulting execution by $\mathcal{E}$. If a process is performing an InsertTask, then we call the treewalk an *i-walk*, otherwise it is an *r-walk*. A complete treewalk requires $O(\log^3 m)$ steps. A step taken by a process performing an i-walk is an *i-step*; otherwise, it is an *r-step*.

Phases. We now provide a framework for bounding the ratio of steps performed versus successful operations during the execution. We begin by splitting the execution into *phases*, which allow us to bound the work performed versus the number of system steps taken. The phase split is different for DoTask and InsertTask operations. Starting from this framework, we analyze the performance the two operations types separately in the next two subsections.

Insert Phases. Consider InsertTask operations. We break the execution into insert phases, defined inductively as follows. The first phase, with index 0, starts with the first i-step; in general, phase number $i > 0$ starts as soon as phase $i-1$ ends. Let s_i be the space at the root at the beginning of phase i, and let $w_i = \min(s_i/4, p)$. Then phase i ends at the point where exactly w_i new InsertTask operations are linearized.[1] We will show that this requires $O(w_i)$ *complete* i-walks to be scheduled during the phase, with high probability. (A treewalk is *complete* in a phase if it begins and ends in that phase.)

Remove Phases. For DoTask operations, the first phase, phase 0, starts with the first r-step in the execution. For $r \geq 1$, the rth remove phase starts as soon as the previous one ends. Let q_r be the number of the processes assigned to DoTask operations at the beginning of the phase, and let u_r be the surplus at the root at the beginning of the phase. Fix $v_r = \min(u_r/4, p)$, and let $\ell_r = \max(q_r, v_r)$. Then the phase ends at the point where exactly v_r new DoTask operations are linearized. We will show that $O(\ell_r)$ complete r-walks are sufficient to move to the next phase, with high probability.

Step Accounting. Given the above phase split, for the purpose of the analysis, we will charge the treewalk steps to phases as follows: for each operation type, steps of treewalks for that operation are counted in the phase in which the corresponding operation is linearized. Steps of unsuccessful treewalks are counted in the phase in which the treewalk reaches the root. The steps by unsuccessful walks which never complete (because of a process crash) will be accounted separately. (There are $O(p \log^3 m)$ such steps in the whole execution.)

[1] Notice that, in the real execution, this might be in the middle of a MaxUpdate operation. The MaxUpdate is not necessarily performed by an i-walk.

4.2.1 DoTask Analysis

In this section, we will consider the execution split into remove phases. Fix a remove phase r as defined above, and let B_r be the set of complete r-walks in that phase. Let u_r be the surplus at the root at the beginning of the phase. By Lemma 1, there exists a set of successful inserts I_r and a set of successful removes D_r such that these operations are the ones counted at the root at the end of this phase, and $|I_r| - |D_r| = u_r$. The set $U_r = I_r \setminus D_r$, is the set of tasks that have been inserted, but not removed, as seen from the root at the beginning of the phase.

Proof Strategy. Our goal is to prove that $\alpha\ell_r$ complete r-walks are sufficient to count $v_r = \min(u_r/4, p)$ new DoTask operations at the root, with high probability. Fix some set of tasks V of size $< v_r$. We will argue that it is very unlikely that *every treewalk* in B_r hits a task in V. By taking a union bound over all such sets V, we will conclude that, with high probability, there is no small set of tasks V which attracts all the treewalks, i.e., the treewalks in B_r have to hit at least v_r distinct tasks.

An important issue is that the set of inserted tasks might grow during this phase, since new i-walks inserting tasks outside U_r may complete concurrently. (For this reason, the set V was not defined to be included in U_r.) This detail does not affect our analysis.

We order the walks in B_r by the time at which they complete their descent. Fix a walk $b \in B_r$. Let e_b be the event that the walk b counts only DoTask operations on tasks in V at the root, and let P_b be the event that all walks that precede b in the order count only DoTask operations in V at the root. Our goal is to bound the probability of the event (e_b *given* P_b), assuming an arbitrary subset of concurrent treewalks. We show that this probability is at most $|V|/u_r$, independent of whether the adversary inserts new tasks.

The bound is obtained in two steps: first, we argue that any set of tasks is likely to be hit by a complete walk (Lemma 3). Then, for a fixed set V, we argue that the set $\overline{V}$, of tasks not in V, is also likely to be hit by each walk (Lemma 4). This yields an upper bound on the probability that all r-walks fall in V (Lemma 5).

Formally, for every node z, let t_z be the linearization time of the MaxScan by the walk b at node z. Define F_z as the set of InsertTask operations in the subtree rooted at z that are counted at z at t_z, and do not have corresponding DoTask operations counted at z at time t_z. Intuitively, the InsertTask operations in F_z constitute the *surplus* at z at t_z. It includes all tasks from U_r in z's subtree that have not been completed, along with any tasks that may have been added since the beginning of the phase. We also define N_z to be the set of tasks inserted in the subtree rooted at z throughout phase r that are *not in* F_z and *not in* V. Intuitively, these newly inserted tasks may be used by the adversary to prevent the walk from reaching tasks in V after reading the count at z. We prove the following:

Lemma 3. *For every node z on a treewalk $b \in B_r$, for a fixed set V, the probability that b starting from node z counts a task in V at the root given that the event P_b did not occur is at least 0 if $F_z \cap V = \emptyset$, and at least $\max((|F_z \cap V| - |N_z|)/|F_z|, 0)$, otherwise.*

Proof. We prove the claim by induction on the height h of the node z. If the node is a leaf, notice that $|F_z| = 1$, since the walk would not have reached the leaf otherwise. Then, the task in F_z is either in V or not in V, which yields the claim. If the height is $h > 0$, then we consider several cases. If $F_z \cap V = \emptyset$, then the probability is trivially at least 0. So $F_z \cap V \neq \emptyset$, therefore there is some task t in V in the subtree rooted at v. If the walk gets stuck at z (since both surpluses at the children are 0), then walk b will count task t at the root, hence the desired probability is 1, and the claim holds. Otherwise, the walk does not get stuck at z.

Let x be the right child of z, and y be the left child of z. Let u_x be the surplus read at x, and u_y the surplus read at y. Assume $u_x > 0$ and $u_y > 0$. (The case where u_x or u_y is 0 follows similarly.) By the inductive hypothesis, the desired probability is at least

$$\frac{u_x}{u_x + u_y} \cdot \frac{|F_x \cap V| - |N_x|}{F_x} + \frac{u_y}{u_x + u_y} \cdot \frac{|F_y \cap V| - |N_y|}{|F_y|}.$$

By definition, $u_x = |F_x|$, and $u_y = |F_y|$. Therefore, the previous lower bound is in fact

$$\frac{|F_x \cap V| + |F_y \cap V| - (|N_x| + |N_y|)}{|F_x| + |F_y|}.$$

Upon close inspection, we notice that $|F_x| + |F_y|$ can be re-written as $|F_z| - |R_{xy}| + |I_V| + |I_{\overline{V}}|$, where 1) R_{xy} is the set of tasks that have been performed and counted at x or at y, but are not counted at z; 2) I_V is the set of $\mathsf{InsertTask}$ operations on tasks in V counted at x or y, but not at z, and 3) $I_{\overline{V}}$ is the set of $\mathsf{InsertTask}$ operations on tasks outside V counted at x or y, but not at z.

If R_{xy} contains a DoTask on a task in V, then the walk will count the operation at the root, and we are done. Therefore, this cannot occur, so $|F_x \cap V| + |F_y \cap V| = |F_z \cap V| + |I_V|$. The previous relation then becomes

$$\frac{|F_z \cap V| + |I_V| - (|N_x| + |N_y|)}{|F_z| - |R_{xy}| + |I_V| + |I_{\overline{V}}|} \geq \frac{|F_z \cap V| - (|N_x| + |N_y| + |I_{\overline{V}}|)}{|F_z| - |R_{xy}|},$$

by simple arithmetic. Finally, notice that, by definition, $N_x \cup N_y \cup I_{\overline{V}} = N_z$. Since $|R_{xy}| > 0$, the resulting lower bound is at least $(|F_z \cap V| - |N_z|)/|F_z|$, as claimed. $\square$

We now relate the surplus F_{root} seen by each walk with the initial surplus U_r, to obtain the following.

Lemma 4. *Given a set V of tasks available in phase r, for every treewalk $b \in B_r$, the probability that b counts a task outside V at the root given that no preceding treewalk counts a task outside V at the root is $\geq 1 - |V|/|U_r|$.*

Proof. Given the set V, let $\overline{V}$ be a set of tasks available in phase r that are not in V. Applying Lemma 3 for the walk b and the set $\overline{V}$ at the root, we have that the probability that $\overline{V}$ is hit by b given that no preceding walk hits $\overline{V}$ is at least $(|F \cap \overline{V}| - |N|)/|F|$, where F is the surplus at the root read by b, and N is the set of inserted tasks from V which had not been counted at the root.

We now relate F and U_r, the initial surplus set. We have that $|F| = |U_r| + |I_{\overline{V}}| + |I_V| - |R_{\overline{V}}| - |R_V|$, where I_V and R_V are the tasks newly inserted and performed from V, respectively, and $I_{\overline{V}}$ and $R_{\overline{V}}$ are the tasks newly inserted and performed from outside V, respectively, as counted at the root upon b's scan. We have that $|R_{\overline{V}}| = 0$; otherwise, a task outside V is counted at the root with probability 1, and we are done. Therefore, $|F \cap \overline{V}| = |\overline{V} \cap U_r| + |I_{\overline{V}}|$. Hence the previous lower bound becomes

$$\frac{|\overline{V} \cap U_r| + |I_{\overline{V}}| - |N|}{|U_r| + |I_{\overline{V}}| + |I_V| - |R_V|} \geq \frac{|\overline{V} \cap U_r| - (|N| + |I_V|)}{|U_r|} = \frac{|U_r| - |V|}{|U_r|},$$

where in the last equality we have used the fact that $V = (V \cap U_r) \cup I_V \cup N$. $\square$

The above claim suggests that there is no benefit for the adversary to insert new tasks during a remove phase, since the probability of hitting a task outside V is always lower bounded by $1 - |V|/|U_r|$. (Recall that $|V| < |U_r|$ throughout.) For simplicity, for the rest of this section, we will assume without loss of generality that no insertions occur during this phase. The next claim leverages the conditional probabilities to obtain a bound on the probability that *all* r-walks are confined in a set V of a given size.

Lemma 5. *For a set of inserted tasks $V \subseteq U_r$, the probability that no treewalk in B_r counts a task outside V as performed at the root by the end of the phase is $\leq (|V|/|U_r|)^{|B_r|}$.*

Proof. First, we observe that the probability that no treewalk in B_r counts a task outside V is:

$$\Pr(e_1 \wedge e_2 \wedge \ldots) \leq \Pr((e_1 \wedge P_1) \wedge (e_2 \wedge P_2) \wedge \ldots).$$

This follows from the fact that P_j is the event that none of the preceding treewalks count a task outside V. By the definition of conditional probability, we know that this is equal to:

$$\Pr(e_1 \wedge P_1) \Pr(e_2 \wedge P_2 | e_1 \wedge P_1) \Pr(e_3 \wedge P_3 | e_1 \wedge P_1, e_2 \wedge P_2) \ldots$$

Finally, from Lemma 4, we know that $\Pr(e_i \wedge P_i | e_1 \wedge P_1, e_2 \wedge P_2, e_3 \wedge P_3, \ldots, e_{i-1} \wedge P_{i-1}) \leq |V|/|U_i|$. We multiply these inequalities to obtain the claim. $\square$

We are now ready to bound the the amount of work duplicated by DoTask operations. Given the previous claim, we upper bound the number of complete walks in a remove phase. We take the union bound over all possible sets V of size $< v_r$ which might contain all walks, and obtain the following.

Lemma 6. *Given $\alpha > 0$ constant, each remove phase $r \geq 0$ contains at most $(\alpha + 3)v_r$ complete r-walks, with probability at least $1 - 1/2^{\alpha v_r}$.*

Proof. Assume there exists a set of $(\alpha + 3)v_r$ complete r-walks in remove phase r, and that less than v_r new tasks are counted at the root. Therefore, there exists a set V of inserted tasks, of size less than v_r, so that all the r-walks in this phase only count tasks in V at the root. By Lemma 5, for a fixed set V with $|V| < v_r$, this probability is less than $(v_r/u_r)^{(\alpha+3)v_r}$, since there are at least $(\alpha + 3)v_r$ walks in the phase. By the union bound, we obtain that the probability that there exists *some* set V with the above property is at most

$$\binom{u_r}{v_r} \left(\frac{v_r}{u_r}\right)^{(\alpha+3)v_r} \leq \left(\frac{u_r e}{v_r}\right)^{v_r} \left(\frac{v_r}{u_r}\right)^{(\alpha+3)v_r} \leq$$

$$\geq e^{v_r} \left(\frac{v_r}{u_r}\right)^{(\alpha+2)v_r} \leq \left(\frac{1}{2}\right)^{\alpha v_r},$$

where in the last step we used the fact that $v_r = \min(p, u_r/4)$. This implies the desired bound. $\square$

Next, we upper bound the total work expended in a remove phase.

Theorem 1. *For each remove phase r, the ratio between the expected number of r-steps counted in the phase and the number of* DoTask *operations linearized in the phase is $O((\ell_r \log^3 m)/v_r)$. The r-steps to operations ratio is $O((\ell_r \log^3 m \log p)/v_r)$ with high probability.*

Proof. We need to prove that the expected number of r-steps in a remove phase r is $O(\ell_r \log^3 m)$. Given initial surplus u_r, the phase ends when $v_r = \min(u_r/4, p)$ new tasks are linearized. By Lemma 6, the expected number of *complete* walks in phase i is $O(v_r)$. By the structure of the algorithm, each of these r-walks costs $O(\log^3 m)$ r-steps.

We then need to bound the number of r-steps contained in other walks counted during this phase. First, we count walks that are linearized in this phase (but do not necessarily start or complete in this phase). The number of such walks is $O(v_r)$ by the definition of the phase.

The extra walks we need to count started in previous phases, but reached the root in phase r, and were unsuccessful. There can be at most one such r-walk for each process assigned to DoTask operations at the beginning of the phase. Therefore, these walks take $O(q_r \log^3 m)$ additional steps. Summing up, we get that the expected r-steps to new operations ratio is $O(\ell_r \log^3 m/v_r)$, as claimed.

For the high probability claim, notice that it holds easily by the above argument if $u_r \geq \log p$. On the other hand, if $u_i < \log p$, then, by Lemma 6, we still have that $\Theta(\ell_r)$ walks will finish the phase with probability at least $1/2$. Therefore, for a costant $c \geq 1$, $c\ell_r \log p$ walks will finish the phase with probability at least $1 - 1/p^c$, as desired. $\square$

We call a remove phase *heavy* if the number of performed operations v_r is at least a constant fraction of q_r, the number of processes performing DoTask operations. In brief, $v_r \geq q_r/k$ for some constant k. The following holds.

Corollary 1. *For each heavy remove phase r, the ratio between the r-steps and the expected successful operations for the phase is $O(\log^3 m)$.*

Unfortunately, we cannot prove a similar statement for all remove phases. The problem lies in the phases that are *not heavy*, when q_r is much larger than v_r. In this case, a lot of processes may compete on only a handful of tasks; since each task can be done by only one process, many processes have to fail and retry. In Section 5, we will prove that this problem is inherent, i.e. *any* algorithm has a similar limitation in this setting.

Finally, we show that every task inserted is eventually performed. The proof is based on the intuition that, given any available task, some process eventually becomes poised to perform it. However, this process might be suspended by the adaptive adversary before performing the task. We argue that, given such a strategy, eventually, *every* process will be poised on the task, which gives the adversary no choice but to allow the task to be counted at the root.

Lemma 7. *The Dynamic To-Do Tree Algorithm ensures that every inserted task is eventually performed.*

Proof. Consider some task x, inserted in phase r_0. (Recall that the task is inserted once it has been counted at the root.) Let $r > r_0$ be a remove phase, and let B_r be the set of walks in r, with $b_r = |B_r|$. It follows by an inductive argument over the height of the tree that there exists an $\epsilon = O(2^{\log^2 m})$ such that, for any phase $r > r_0$ at the beginning of which task x has not been performed, the probability that no r-walk in B_r reaches the leaf associated with x is at most $(1-\epsilon)^{b_r}$. Note that the adversary could suspend every process that is poised to perform x as soon as it reaches the leaf. However, by the above argument, it follows that, in an infinite execution, with probability 1, every non-failed process will be eventually poised to perform task x. Therefore the task is eventually performed, even though the completion time may be high because of the adaptive adversary. □

4.2.2 Insert Analysis In this section, we lower bound the performance of insert operations. For the rest of the section, we will consider the execution split into insert phases. Fix an insert phase i as defined in Section 4.2, and let B_i be the set of complete i-walks in the phase. Let s_i be the *space* at the root at the beginning of the phase. By Lemma 1, there exists a set T of insert operations, and a set D of remove operations such that $m - s_i = |T| - |D|$. We say that a leaf is *free at the root* at some time t if, given the tasks counted at the root at t, each insert task on the leaf counted at the root has a matching remove task on the leaf counted at the root. (The property can also hold vacuously.) It follows that, at the beginning of the phase, there exist at least s_i distinct leaves that are free at the root. Let S_i be this set of leaves.

We say that an successful insert is counted at the root during this phase if it is propagated to the root by some walk during this phase.

The argument is similar to the one for remove operations: we aim to bound the probability of the event $\mathcal{D}$ that more than $(\alpha+3)w_i$ walks complete in the phase without inserting w_i new tasks. In other words, we wish to upper bound the probability that $\Theta(w_i)$ i-walks in the phase only hit leaves from a subset of size w_i, out of the total of at least $s_i \geq 4w_i$ available spots.

First, notice that the initial space s_i at the root may in fact increase during the phase, as some tasks are removed by concurrent DoTask operations. We will show that this does not affect the analysis significantly.

We also notice a subtle technical issue: by the structure of the algorithm, the set of successful InsertTask operations at a certain leaf can propagate up the tree *ahead* of the set of removes at the same leaf. (See Figure 2 for an illustration.) In particular, two consecutive inserts at a leaf can be counted at an ancestor node *before* the intermediate DoTask operation. Consequently, the space at a node v may be *smaller* than the size of the set of leaves that are free in the subtree rooted at v.

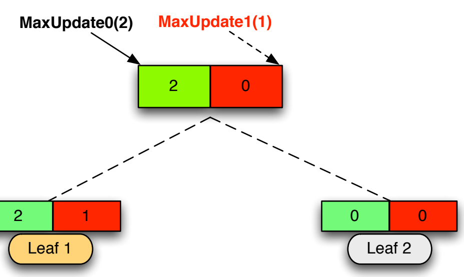

Figure 2: Example of asymmetric propagation. The second insert on Leaf 1 first updates the insert count to 2, and then is stalled before updating the remove count to 1. A concurrent insert walk would see 0 space in the subtree with two leaves, even though Leaf 2 always has space 1. The *bias* in this subtree therefore has value 1.

The immediate consequence for our analysis is that the adversary may bias the probability of walks $b \in B_i$ hitting V in two ways: by decreasing the space at certain internal node, therefore making it less likely that the walks will hit certain sets V, and by making extra leaves available outside V. Our strategy will be to bound the probability bias that the adversary can introduce through these two procedures. For the first issue, we will show that the adversary can only generate bias by allowing walks to count extra operations at the root; therefore, the existence of bias can affect the probability of hitting a certain set, but not that of hitting a certain number of leaves.

Node Overflow and Bias. For this, we define the *overflow* at a node v of height h as follows. Let s_v be the space at a node v, as read by some walk b. By Lemma 1, there exists a set of insert operations I_v and a set of DoTask operations D_v, such that exactly those operations are counted at v, and $|I_v| - |D_v| =$

$2^{height(v)} - s_v$. From Lemma 1, s_v is at most $2^{height(v)}$. On the other hand, notice that the value of the surplus is always bounded in the algorithm to be at most $2^{height(v)}$; therefore, in the following, we consider that the space at a node is always ≥ 0. (While the space can be negative at a node because of the issues outlined above, it does not change the analysis.)

Let T_ℓ be the InsertTask operations on leaf ℓ that are counted at v, while a corresponding DoTask operation on the same task is not counted at v. Notice that having one such insert for each leaf in the subtree is normal—this is an insert without its corresponding remove. However, by the structure of the algorithm, it is possible to have *several* such inserts for the same leaf, which propagate ahead of their corresponding removes. Let O_v be the set of such operations at a node v, which is formed by taking the union of the sets T_ℓ for each descendant leaf ℓ of v, from which we remove the first insert operation on that leaf, if such an operation exists. We define the overflow at v to be O_v.

The *new overflow* for phase i at some node v, denoted by N_v^i, is the overflow at v that is a consequence of InsertTask operations *not counted* at the root at the beginning of phase i. Finally, we define the *bias* at some node v in phase i, Q_v^i as the union of all new overflow sets N_w^i, where w is a descendant node of v, including v itself.

Formally, fix a set of leaves V of size $< w_i$. We sort the walks in B_i by the order in which they complete their descent. Fix a walk $b \in B_i$, and let e_b be the event that b counts only operations on leaves in V at the root, and P_b be the event that all preceding walks only count operations on leaves in V at the root. We aim to bound the probability of the event e_b given P_b.

Again, we obtain the bound by induction. Given node z in the tree, let t_z be the linearization point of the MaxScan of b at z. Let E_z be the set of leaves which are free at z at time t_z: these are leaves on which every InsertTask counted at z can be paired with a DoTask counted at z. Intuitively, E_z constitutes the *space* at z when the walk b accesses z. We also define R_z to be the set of leaves in the subtree rooted at z that become free during phase i, but are *not in* E_z and *not in* V. Intuitively, the adversary can use these extra leaves to bias the walk b away from V after it has scanned the count at z. We also define the set of tasks Q_z to be the bias at z, in phase i. We claim the following lower bound on the probability that b hits V.

Lemma 8. *For every node z on a treewalk $b \in B_i$, the probability that a treewalk b starting from node z counts a leaf in the set V given event P_b is at least* 0 *if* $E_z \cap V = \emptyset$, *and at least* $\max((|E_z \cap V| - |Q_z| - |R_z|)/|E_z|, 0)$ *otherwise.*

Proof. We proceed by induction on h, the height of node z. If z is a leaf, then the claim follows: the overflow at a leaf is always 0, and the leaf is either in V or it is not. If the height of the node is $h > 0$, then we consider several cases. If $E_z \cap V = \emptyset$, then the claim is trivial. So $|E_z \cap V| > 0$, therefore there exists some set $L \subseteq V$ of leaves in the subtree rooted at V. If the walk gets stuck at z, then either some insert on a leaf in L is counted at z, or there must exist at least $|L|$ bias on a set of leaves not in L, which are descendants of z. In both cases, the claim holds.

Otherwise, the walk does not get stuck at z. Let x be the left child of z, and y be the right child. Let s_x be the space read at x, and s_y be the space read at y. Assume $s_x > 0$ and $s_y > 0$. (The case when one of them is 0 follows similarly.) By the inductive hypothesis, the probability of hitting V is at least

$$\frac{s_x}{s_x + s_y} \frac{|E_x \cap V| - |Q_x| - |R_x|}{|E_x|} + \frac{s_y}{s_x + s_y} \frac{|E_y \cap V| - |Q_y| - |R_y|}{|E_y|} = \frac{|E_x \cap V| + |E_y \cap V| - (|Q_x| + |Q_y|) - (|R_x| + |R_y|)}{|E_x| + |E_y|},$$

since, by definition, $s_x = |E_x|$, and $s_y = |E_y|$. We can now write $|E_x| + |E_y| = |E_z| + |A_V| + |A_{\overline{V}}| - (|I_V| + |I_{\overline{V}}|)$, where 1) A_V and $A_{\overline{V}}$ are space additions over E_z on leaves in V and outside V, respectively, caused by DoTask operations which are counted at x or y, but not at z, and 2) I_V and $I_{\overline{V}}$ are space removals over the space in E_z on leaves in V and outside V, respectively, caused by InsertTask operations which are counted at x or y, but not at z. Notice that if $I_V \neq \emptyset$, then the walk b will mark some insert operation on a leaf in V at the root, which proves the claim. Therefore, $|I_V| = 0$. At the same time, this implies that $|E_x \cap V| + |E_y \cap V| = |E_z| + |A_V|$. We obtain that the desired probability is at least

$$\frac{|E_z \cap V| + |A_V| - (|Q_x| + |Q_y|) - (|R_x| + |R_y|)}{|E_z| + |A_V| + |A_{\overline{V}}| - (|I_V| + |I_{\overline{V}}|)} \geq \frac{|E_z \cap V| - (|Q_x| + |Q_y|) - (|R_x| + |R_y| - |A_{\overline{V}}|)}{|E_z|} = \frac{|E_z \cap V| - |Q_z| - |R_z|}{|E_z|},$$

where in the last step we have used that $|R_z| = |R_x| + |R_y| + |A_{\overline{V}}|$ and $|Q_z| = |Q_x| + |Q_y|$, according to the definitions. This concludes the proof. $\square$

Let Q be the bias at the root in this phase. We obtain the following probability bound for single treewalks.

Lemma 9. *For every treewalk $b \in B_i$, the probability that b counts an insert on a leaf* outside V *at the root given that every preceding preceding treewalk counts inserts on a leaf in V at the root is $\geq 1-(|V|-|Q|)/|S_i|$.*

Proof. Given the set V, let $\overline{V}$ be the set of free leaves in phase i that are not in V. Applying Lemma 8 to the set $\overline{V}$ and the walk b at the root, we obtain that the probability that $\overline{V}$ gets hit by b given that no preceding walk hits $\overline{V}$ is at least $(|E \cap \overline{V}| - |R|)/|E|$, where E is the space at the root read by b, and R is the set of leaves in V which were not counted at the root when b scanned.

We now relate E and the initial surplus set S_i. We have $|E| = |S_i| - |I_V| - |I_{\overline{V}}| + |R_V| + |R_{\overline{V}}|$, where I_V and R_V are the tasks newly inserted and performed on leaves in V, respectively, and $I_{\overline{V}}$ and $R_{\overline{V}}$ are the tasks newly inserted and performed on leaves outside V, respectively, as counted at the root upon b's scan. If $|I_{\overline{V}}| > 0$, then the probability is 1, and the claim holds. Therefore, $|E \cap V| = |S_i \cap V| + |R_{\overline{V}}|$. Hence the bound becomes

$$\frac{|S_i \cap \overline{V}| + |R_{\overline{V}}| - |R| - |Q|}{|S_i| - |I_V| - +|R_V| + |R_{\overline{V}}|} \geq$$

$$\frac{|S_i \cap \overline{V}| - |R| - |Q| - |R_V|}{|S_i|} = \frac{|S_i| - |V| - |Q|}{|S_i|},$$

where we have used that, by definition, $|V| = (|S_i| - |S_i \cap \overline{V}|) + |R| + |R_V|$ in the last step. □

Notice that the previous claim proves that the adversary can derive no benefit from scheduling new DoTask operations during the insert phase in terms of confining all walks to the set V. For simplicity, for the rest of this section we shall assume without loss of generality that no tasks get removed during this phase. We put these results together to obtain a bound on the probability of hitting a set V of a given size.

Lemma 10. *For a set of leaves $V \subseteq S_i$, the probability that no treewalk in B_i counts an insert at a leaf outside V is $\leq ((|V| - |Q|)/|S_i|)^{|B_i|}$.*

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

$$\Pr(e_1 \wedge e_2 \wedge \ldots) \leq \Pr((e_1 \wedge P_1) \wedge (e_2 \wedge P_2) \wedge \ldots).$$

This follows from the fact that P_j is exactly the event that no preceding treewalk counts an insert at a leaf outside V. By the definition of conditional probability, this is equal to:

$$\Pr(e_1 \wedge P_1) \cdot \Pr(e_2 \wedge P_2 | e_1 \wedge P_1) \cdot \Pr(e_3 \wedge P_3 | e_1 \wedge P_1, e_2 \wedge P_2) \cdot \ldots$$

Finally, from Lemma 9, $\Pr(e_i \wedge P_i | e_1 \wedge P_1, e_2 \wedge P_2, e_3 \wedge P_3, \ldots, e_{i-1} \wedge P_{i-1}) \leq (|V| - |Q|)/|S_i|$, which concludes the proof. □

We are now ready to analyze the performance of InsertTask operations. Recall that $w_i = \min(p, s_i/4)$. We prove the following.

Lemma 11. *For $\alpha > 0$ constant, each insert phase $i \geq 0$ as defined above contains at most $(\alpha + 3)w_i$ complete i-walks, with probability at least $1 - 1/2^{\alpha w_i}$.*

Proof. Assume there exists a set of $(\alpha + 3)w_i$ complete i-walks scheduled in phase i, and that less than w_i new insert tasks are counted at the root by the end of the last complete walk. Therefore, there exists a set V of size less than w_i so that all the i-walks in this phase only count tasks at leaves in this set at the root. Assume $s_i/4 \leq p$, so $w_i = s_i/4$. (The case $s_i/4 > p$ is similar.)

Let Q be the bias set for the tree root in this phase. Notice that, by construction, none of the tasks in the bias set were counted at the root at the beginning of phase i, and all of them will be counted at the root by the end of the phase. Notice that this automatically implies that the size of the bias Q at the root is at most $s_i/4$.

Second, since all the walks are concentrated in a set of size at most $s_i/4$, there must exist a set $V \subset S_i$ of leaves of size at least $3s_i/4$ such that no walk in B_i hits a leaf in V. By Lemma 10, we obtain that the probability that such a set V exists is at most

$$\binom{s_i}{3s_i/4}\left(1 - \left(\frac{3}{4} - \frac{1}{4}\right)\right)^{(\alpha+3)w_i} \leq$$

$$\left(\frac{s_i e}{s_i/4}\right)^{s_i/4}\left(\frac{1}{2}\right)^{(\alpha+3)w_i} \leq e^{s_i/4}\left(\frac{1}{2}\right)^{(\alpha+2)w_i} \leq$$

$$\left(\frac{1}{2}\right)^{\alpha w_i},$$

which implies the desired bound. □

This implies the following.

Lemma 12. *For each insert phase i, the ratio between the expected number of i-steps counted in the phase and the number of* InsertTask *operations linearized in the phase is $O(\log^3 m)$. The i-steps to linearized insert operations ratio is $O(\log^3 m \log p)$ with high probability.*

Proof. We need to prove that the expected number of i-steps in some insert phase i is $O(w_i \log^3 m)$. Given initial space s_i, the phase ends when w_i new tasks are linearized. By Lemma 11 and the properties of the geometric distribution, the expected number of *complete* walks in phase i is $O(w_i)$. By the structure of the algorithm, each of these i-walks costs $O(\log^3 m)$ i-steps.

We then need to bound the number of i-steps contained in other walks counted during this phase. First, we count walks that start and are linearized in this phase, but do not complete in this phase. The number of such walks is $O(w_i)$ by the definition of the phase.

The extra walks we need to count started in previous phases, but reached the root in phase i, or were linearized in phase i. By the task semantics, for TaskInsert operations there can be at most $\min(s_i, p)$ distinct such walks: otherwise, at the beginning of the phase, the adversary must have scheduled $> s_i$ distinct inserts into s_i space, a contradiction. Therefore, the additional cost for these walks is $O(w_i \log^3 m)$ steps. Summing up, we get that the i-steps / new operations ratio is $O(w_i \log^3 m / w_i) = O(\log^3 m)$.

For the high probability claim, notice that it holds easily by the above argument if $w_i \geq \log p$. On the other hand, if $w_i < \log p$, then, by Lemma 11, we still have that $\Theta(w_i)$ walks will finish the phase with probability at least 1/2. Therefore, for a costant $c \geq 1$, $cw_i \log p$ walks will finish the phase with probability at least $1 - 1/p^c$, as desired. □

This implies the final performance claim.

Theorem 2. *Consider an input I, and assume a time t during an execution of the Dynamic To-Do Tree algorithm on I. For $x \geq p/2$, let $T_i(t, x)$ be the number of i-steps the algorithm requires to perform x inserts starting from time t. Then we have that $E[T_i(t, x)] = O(x \log^3 m)$, and that $T_i(t, x) = O(x \log^3 m \log p)$, with high probability.*

Proof. Let t' be the time when x new insert operations are performed, starting from time t. (In theory, t' could be infinite.) We consider the interval $[t, t']$ split into insert phases. Let i be the phase which contains time t, and let $j \geq i$ be the phase which contains time t'. By the definition of a phase, phases i and j perform at most p new inserts. By Lemma 12, each is charged at most $O(p \log^3 m)$ total steps in expectation.

Each intermediate phase $k \in \{i+1, \ldots, j-1\}$ performs some w_k work, being charged expected $O(w_k \log^3 m)$ steps in total, with $\sum_k w_k \leq x$. Therefore, the total expected number of steps charged during the interval $[t, t']$ is in $O((p + x) \log^3 m) = O(x \log^3 m)$, as claimed. The high probability claim follows identically. □

5 Competitive Analysis

Finally, we compare the performance of our algorithm with that of an optimal algorithm OPT for the dynamic task allocation problem. We prove the following claim.

Theorem 3. *For any input I, let $W(I)$ be the expected worst-case cost of our algorithm under input I, and let $W(OPT, I)$ be the expected worst-case cost of the optimal algorithm under I. Then $W(I) \leq c\, W(OPT, I) \log^3 m$, for a constant c.*

Proof Structure. We fix an input I, containing n operations. Let n_i be the number of inserts in I, and n_r be the number of removes in I. Fix A to be a worst-case adversarial strategy for our algorithm. Let $\mathcal{E}$ be an execution of the algorithm under A, and let $C(\mathcal{E}, I)$ be the total work in $\mathcal{E}$. The expected worst-case cost of our algorithm under I is $W(I) = E[C(\mathcal{E}, I)]$, where the expectation is taken over processes' coin flips.

Our argument bounds the number of i-steps and r-steps in the execution separately. The number of i-steps is $O(n_i \log^3 m)$, by Theorem 2. To bound r-steps, we need to take into account the number of processes that may be poised to perform DoTask operations at the beginning of each phase.

Preliminaries. For the rest of this section, let I be a fixed input. Let n_i be the number of InsertTask operations, and n_r be the number of DoTask operations in the input. Also, let $n = n_i + n_r$. Let A be the worst-case adversarial strategy for our algorithm. Let $\mathcal{E}$ be an execution of the algorithm under A, and let $C(\mathcal{E}, I)$ be the total work in $\mathcal{E}$. The expected worst-case cost of our algorithm under I is $W(I) = E[C(\mathcal{E}, I)]$, where the expectation is taken over processes' coin flips. For accounting purposes, we consider the number of steps spent by processes on each operation type separately. Let $C(\mathcal{E}, I)_{ins}$ be the number of steps spent on InsertTask operations, and let $C(\mathcal{E}, I)_{do}$ be the number of steps spent on DoTask operations in an execution $\mathcal{E}$. Obviously, $C(\mathcal{E}, I) = C(\mathcal{E}, I)_{ins} + C(\mathcal{E}, I)_{do}$.

Bounding Insert Steps. We first upper bound the number of i-steps in the execution. The claim follows directly from Theorem 2, by summing up the total number of *i-steps* across phases.

Claim 1. *For some constant c, $E[C(\mathcal{E}, I)_{ins}] \leq c\, n_i \log^3 m$.*

Bounding Remove Steps. Consider the split of execution $\mathcal{E}$ into remove phases, as described in Section 4.2. Fix a remove phase r. Let q_r be the number of the processes assigned to DoTask operations in the beginning of the phase, and let u_r be the corresponding surplus. Let $v_r = \min(u_r/4, p)$, and let $\ell_r = \max(q_r, v_r)$. The phase contains exactly v_r new DoTask operations.

Theorem 1 suggests that the number of r-steps in a phase r is proportional to $\max(q_r, v_r)$. To bound the total number of steps, we need to upper bound this quantity for each phase. Fix a phase r, and consider the parameters (q_r, v_r) at its beginning, and their values (q_{r+1}, v_{r+1}) at the beginning of the next phase. The value of v_{r+1} is uniquely determined by p and the surplus u_r. Thus, to maximize the expected cost of the algorithm, it is in the adversary's interest to maximize q_{r+1}, the number of processes performing DoTask operations at the beginning of the phase. We now define a way to upper bound the value of $\sum_r q_r$ over the course of an execution.

We define a *pattern* to be a sequence of operation types $(\mathsf{InsertTask}, \mathsf{Dotask}, \ldots)$. Notice that each execution induces a pattern as output: the linearization order of performed operations. Given an initial state and an input, the output pattern of an execution determines the insert and remove phase split of the execution: the beginning of each phase is the end of the next (or the first step in the execution), and the end of the phase is defined as the point where some number of new operations have completed. (Recall that upon completing an operation, a process gets the next available input task.) Importantly, the pattern and the input determine the tuples $(q_r, v_r)_r$ for every remove phase $r \geq 1$, since the pattern determines the order in which operations are completed, and processes are then assigned the next available operation in the input. The output pattern of an execution is controlled by the adversary, since it controls the point when a process is scheduled to complete its current operation. For a pattern π, let $f(\pi)$ be the sum $\sum_{r \geq 1} q_r$. Notice that not every pattern can match a certain input. We say that π *agrees* with the input I, $\pi \models I$, if there exists an execution of our algorithm against the adversary A under input I matching π. For fixed input I, let $\pi_{\max}$ be the pattern with $f(\pi_{\max}) = \max_{\pi \models I} f(\pi)$. Obviously, $\pi_{\max}$ agrees with I. The following claim encapsulates a few properties of patterns.

Claim 2. *Given a fixed number of tasks inserted initially, the input I and the pattern π uniquely determine the phase split of the execution and the values (q_r, v_r) for every phase r. Given any pattern $\pi \models I$, the adversary can induce any execution $\mathcal{E}$ on I to produce pattern π.*

Proof. Fix an input I and a pattern π. Assume an arbitrary number of tasks inserted in the data structure initially. This gives an initial surplus u_0, and an initial space s_0 for the execution. Upon wakeup, each process gets assigned the next available task in the input. This gives the initial number of processes q_0 that are assigned DoTask operations. Tasks get completed in the order specified by the pattern π. Once a process completes a task, it gets the next task from the input. Therefore the number of processes assigned to DoTask operations is entirely determined after each new operation in the pattern completes. The end of each phase i is specified by the pattern (we count up the point when enough new operations complete). By the above argument, the number of processes poised on DoTask operations at the end of the phase is also determined at the end of each phase by the input and by the pattern.

To see that the adversary controls the pattern, first recall that each DoTask operation must contain a TryTask, and each InsertTask operation must contain a PutTask. Notice that, to control the pattern, it is enough for the adversary to control the type of the next operation performed. For example, to ensure that the next operation in the pattern is an insert, the adversary may simply suspend all active DoTask operations before performing their TryTask call. As long as some process is currently assigned to an insert (necessary for the adversary's target pattern to agree with I), this process can be scheduled to complete next. Ensuring that the next operation is a remove is symmetric. □

We can now upper bound the expected number of remove steps in an execution of our algorithm as a function of the number of remove operations and $f(\pi_{\max})$.

Claim 3. *For some constant $c \geq 1$, $E[C(\mathcal{E}, I)_{do}] \leq c(n_r + f(\pi_{\max})) \log^3 m$.*

Proof. By Theorem 1, the total expected number of steps in the execution is upper bounded by $\sum_r \alpha(v_r + q_r) \log^3 m = \sum_r \alpha v_r \log^3 m + \sum_r \alpha q_r \log^3 m$, where the sums are taken over all the remove phases. The first sum is at most $\alpha n_r \log^3 m$, while the second term is at most $\alpha f(\pi_{\max}) \log^3 m$, by definition. This implies the claim. □

The following lemma provides the last missing piece in the proof of Theorem 3 by establishing a lower bound on the worst-case cost of the optimal algorithm.

Lemma 13. $W(OPT, I) \geq \max(n, f(\pi_{\max})/2)$.

Proof. The optimal algorithm has to perform n operations, therefore at least n steps. To prove the lower bound of $f(\pi')$ steps, let us define the adversarial strategy and the adversary A' that ensures that every execution of the optimal algorithm OPT follows the pattern $\pi_{\max}$. As $\pi_{\max}$ agrees with I, there exists an execution $\mathcal{E}'$ of our algorithm against A under I that matches $\pi_{\max}$. The strategy of A' is built on $\mathcal{E}'$ as follows:

- When an available process is assigned to a new operation from I in $\mathcal{E}'$, A' assigns an available process to a new operation of the same type in the execution of OPT.
- When a process takes a step but does not complete the operation in $\mathcal{E}'$, no steps are taken in OPT's execution.

- When a process completes an operation in $\mathcal{E}'$ and becomes available, the adversary A' first lets each process assigned to an operation of the same type reach a state where the next step would complete its operation. If the operation is InsertTask, A' lets one of these processes complete the operation. If it is DoTask, then consider a task that would be done by the *highest number* of processes if they were to take the next step. Strategy A' just lets all these processes take the next step. As tasks can be done only once, only one process will be successful.

The fact that this is a valid strategy is proved by the following Lemma.

Lemma 14. *Consider the execution $\mathcal{E}'$ step-by-step. After each step, let the corresponding steps, defined by the adversarial strategy, complete in the execution of OPT. At all these points, where the corresponding steps complete, the number of available processes, the number of processes assigned to* InsertTask *and the number of processes assigned to* DoTask *operations are the same in both executions. The type of an operation from I that will be assigned next is also the same in both executions.*

Proof. Define the state of an execution as a tuple (a, t, q, i), where a is the number of the available processes, t is the number of processes assigned to InsertTask operations , q is the number of processes assigned to the DoTask operations, with $a+t+q=p$ and $i \geq 0$ is the index of the next operation to be assigned in input I. Let $\mathcal{E}'_{opt}$ denote the current execution of OPT against the adversarial strategy. The basis of the induction holds since the initial state for both executions is $(p, 0, 0, 0)$: p processes are available and the next operation to be assigned is the first operation in I. For the induction step, assume that both $\mathcal{E}'$ and $\mathcal{E}'_{opt}$ are in state (a, t, q, i), and a step is performed in $\mathcal{E}'$. We have one of the following cases.

- If the step assigned an available process to i-th operation from I, there had to be an available process, so $a \geq 1$ holds. Then, because the state of $\mathcal{E}'_{opt}$ was the same, there are available processes in $\mathcal{E}'_{opt}$ and the i-th operation of I will be assigned to one of them.

  By construction, the i-th operation from I was assigned in both $\mathcal{E}'$ and $\mathcal{E}'_{opt}$, therefore the next operation assigned will have index $i+1$. If the i-th operation was a InsertTask, a new state will be $(a-1, t+1, q, i+1)$, and if the operation was a DoTask then the new state will be $(a-1, t, q+1, i+1)$ in both executions and the inductive step holds.

- If a process already assigned to an operation in $\mathcal{E}'$ took a step, a state of the execution $\mathcal{E}'$ would remain unchanged. By construction, no step is taken in $\mathcal{E}'_{opt}$ and this also does not change the state of $\mathcal{E}'_{opt}$. $\mathcal{E}'$ and $\mathcal{E}'_{opt}$ were in the same state and they stay in the same state.

- If a process completed an operation and became available in $\mathcal{E}'$, depending on the type of the operation completed, the new state in $\mathcal{E}'$ becomes $(a+1, t-1, q, i)$ or $(a+1, t, q-1, i)$. In $\mathcal{E}'_{opt}$, initially some steps are performed that do not change the state (bringing all the processes to the point where they are about to complete). In the end, by the above construction and argument, only a single process completes an operation successfully in $\mathcal{E}'_{opt}$, and the type of operation is the same as the type of operation completed in $\mathcal{E}'$, therefore the final state in $\mathcal{E}'_{opt}$ again matches the state of $\mathcal{E}'$.

□

We have that $\mathcal{E}'_{opt}$ is any execution of OPT under input I against the adversary A'. If a next step of a process completes a task, the process is said to be *covering* the task. The way A' schedules DoTask operations in $\mathcal{E}'_{opt}$ according to the above defined strategy, is that the operation with the most processes covering it finishes first. In any phase with parameters s and q, performed $s/2$ DoTask operations would have to be covered by more processes than the other $s/2$ tasks, so they would be covered by at least $q/2$ processes. All of these processes would take a step and therefore, the optimal algorithm OPT has to perform at least $q/2$ steps. Moreover, $\mathcal{E}_{opt}$ matches $\pi_{\max}$, so the phases are determined by $\pi_{\max}$ and the sum of $q/2$ values for all phases is $f(\pi_{\max})/2$. The adversary A' ensures that OPT does at least $f(\pi_{\max})/2$ work in every execution, so the worst-case work is at least $f(\pi_{\max})/2$. The proof is complete. □

Since $W(I) = E[C(\mathcal{E}, I)] = E[C(\mathcal{E}, I)_{ins}] + E[C(\mathcal{E}, I)_{do}]$ the theorem follows by putting together the previous claims.

6 Conclusions and Future Work

We have presented the first algorithm for the dynamic task allocation problem, which is within logarithmic factors of optimal. Our results show that, using randomization, processes can cooperate to share work efficiently even in strongly adversarial conditions. Interesting directions for future work would be to explore the practical implications of our results for long-lived data structures, and to see our algorithm can be adapted to obtain long-lived solutions for other problems such as renaming [7] or distributed counting.

7 Acknowledgements

The authors would like to thank Nir Shavit for useful discussions and support, and the anonymous reviewers for their insightful comments.

References

[1] Yehuda Afek, Michael Hakimi, and Adam Morrison. Fast and scalable rendezvousing. In *DISC*, pages 16–31, 2011.

[2] Yehuda Afek, Guy Korland, Maria Natanzon, and Nir Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Euro-Par (2)*, pages 151–162, 2010.

[3] Miklós Ajtai, James Aspnes, Cynthia Dwork, and Orli Waarts. A theory of competitive analysis for distributed algorithms. In *Proc. 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 401–411, 1994.

[4] Dan Alistarh, Michael A. Bender, Seth Gilbert, and Rachid Guerraoui. How to allocate tasks asynchronously. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 331–340, 2012.

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

[6] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Faster than optimal snapshots (for a while). In *2012 ACM Symposium on Principles of Distributed Computing*, pages 375–384, July 2012.

[7] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Ruediger Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.

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

[9] Dmitry Basin, Rui Fan, Idit Keidar, Ofer Kiselov, and Dmitri Perelman. Café: Scalable task pools with adjustable fairness and contention. In *DISC*, pages 475–488, 2011.

[10] Michael A. Bender and Seth 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] Jonathan F. Buss, Paris C. Kanellakis, Prabhakar L. Ragde, and Alex Allister Shvartsman. Parallel algorithms with processor failures and delays. *J. Algorithms*, 20:45–86, January 1996.

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

[13] Bogdan S. Chlebus, Roberto De Prisco, and Alexander A. Shvartsman. Performing tasks on synchronous restartable message-passing processors. *Distributed Computing*, 14(1):49–64, 2001.

[14] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: scalable nonzero indicators. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 13–22, New York, NY, USA, 2007. ACM.

[15] Chryssis Georgiou and Dariusz R. Kowalski. Performing dynamically injected tasks on processes prone to crashes and restarts. In *DISC*, pages 165–180, 2011.

[16] Chryssis Georgiou, Alexander Russell, and Alexander A. Shvartsman. The complexity of synchronous iterative do-all with crashes. *Distributed Computing*, 17(1):47–63, 2004.

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

[18] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

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

[20] Sotiris Kentros, Chadi Kari, and Aggelos Kiayias. The strong at-most-once problem. In *DISC*, pages 386–400, 2012.

[21] Sotiris Kentros, Aggelos Kiayias, Nicolas C. Nicolaou, and Alexander A. Shvartsman. At-most-once semantics in asynchronous shared memory. In *DISC*, pages 258–273, 2009.

[22] Dariusz R. Kowalski and Alexander 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] Grzegorz 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.

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

[25] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks. *Theory Comput. Syst.*, 30(6):645–670, 1997.

[26] Nir Shavit and Asaph Zemach. Combining funnels: A dynamic approach to software combining. *J. Parallel Distrib. Comput.*, 60(11):1355–1387, 2000.

A An Unbounded MaxArray Implementation

The unbounded MaxArray, whose pseudocode is given in Algorithm 1, uses a CAS object[2] C, and an array of bounded MaxArrays MA, built using the construction of Aspnes et al. [6]. We consider their maximum values to be $H = K = p^\alpha$, for $\alpha > \beta > 1$ constant, where we

[2] We assume a CAS object supports a read and a compare-and-swap operation, with the usual semantics.

Shared:
Register $C = (V_0, V_1, P)$
Vector of MaxArrays MA, with maximum values $H = K = p^\alpha$
procedure MaxScan()
  $(V_0, V_1, P) \leftarrow C.\mathsf{read}()$
  $(v_0, v_1) \leftarrow MA[P].\mathsf{MaxScan}()$
  return $(V_0 + v_0, V_1 + v_1)$
procedure MaxUpdate0(v)
  $(V_0, V_1, P) \leftarrow C.\mathsf{read}()$
  $v' \leftarrow v - V_0$
  if $v' \leq 0$ **then**
    return *success*
  if $v' \leq h$ **then**
    $MA[P].\mathsf{MaxUpdate0}(v')$
    if $C.\mathsf{read}() = (V_0, V_1, P)$ **then**
      return *success*
    else
      MaxUpdate0(v)
  else
    $(v_0, v_1) \leftarrow MA[P].\mathsf{MaxScan}()$
    $(u_1, u_2, u_3) \leftarrow C.\mathsf{CAS}((V_0 + v_0, V_1 + v_1, P), (V_0, V_1, P))$
    MaxUpdate0(v)

Algorithm 1: The Unbounded MaxArray algorithm.

have previously fixed $m = p^\beta$. The CAS register has three sub-fields: V_0, the remove offset, V_1, the insert offset, and P, the index of the current active MaxArray. All these fields are read and updated at the same time.

The intuition behind the data structure is that we use each of the MaxArray objects to store values up to their maximum capacity; when this is exceeded, we store the extra count in the CAS object as an offset, and change the pointer to the next object in MA.

More precisely, to *read* the unbounded MaxArray, a process reads the CAS to get the current offsets (V_0, V_1) and the pointer to the current active MaxArray in MA. The process then snapshots the current value (v_0, v_1) in the bounded MaxArray, and returns the sum $(V_0 + v_0, V_1 + v_1)$.

If a process needs to update the first cell (insert count) of the unbounded MaxArray to some value v, it proceeds as follows (updating the remove count is symmetric). The process first reads the CAS C to get (V_0, V_1, P), then computes the value $v' = v - V_0$ that it should write to the current active MaxArray in MA, pointed to by P. If $v' \leq 0$, the process simply returns. If $v' > 0$ but is smaller than the maximum value h for the corresponding cell of the MaxArray, then the process writes the value to the current MaxArray using a MaxUpdate0 operation. Finally, if the value is *larger* than the maximum value of the MaxArray, then the process attempts to update the current value of C to $(V_0 + v_0, V_1 + v_1, P + 1)$, i.e. increasing the value offsets and moving up the MA array index by one. It then calls MaxUpdate on the modified object to complete its operation.

Analysis. The safety of the algorithm is straightforward to prove, and reads are wait-free. The performance of the algorithm is formalized by the following claim.

Lemma 15. *Given the unbounded* MaxArray *in Algorithm 1, where increments are bounded by p, the amortized step complexity of a* MaxScan *operation is* $O(\log^2 p)$*, and the amortized step complexity of a* MaxUpdate *is* $O(\log p)$*, assuming that a* CAS *operation costs a constant number of steps, and that* $H = K = p^\alpha$ *for* $\alpha \geq 2$ *constant.*

Proof. We define an *epoch* $i \geq 0$ to be the interval between successful[3] CAS operations number i and $i + 1$. We now consider the unbounded MaxArray in the context of the dynamic to-do tree, and lower bound the number of MaxUpdate operations in each epoch to be at least polynomial in p.

Consider a MaxUpdate(v) operation by process p that causes the process to invoke a CAS operation. This implies that $v - V_0$ is at least $H \geq p^\alpha$, for $\alpha \geq 2$ constant. On the other hand, notice that, by Lemma 1, in the dynamic to-do tree, the maximum difference between the value v that a process is updating and the value currently in the unbounded MaxArray is p: there can be at most p distinct walks suspended in the subtree corresponding to the current MaxArray. This implies that, whenever some process wants to write v to the unbounded MaxArray, the current value of the MaxArray is at least $v - p$. Hence the new offset value that the process is proposing to the CAS is at least $V_0 + p^\alpha - p$, therefore polynomially many operations are taken into account in the epoch. (Also, the difference between the two indices of the MaxArray can be at most $m = p^\beta$, for $\alpha > \beta > 0$; therefore, there also exist at least $p^\alpha - p^\beta - p$ MaxUpdate1 operations that succeeded in the epoch.) Hence, there are polynomially many MaxUpdate operations on each MaxArray index in an epoch. On the other hand, the number of CAS operations and restarted MaxUpdate operations for each epoch change is at most p (since each process may be in only one of these categories), therefore we can amortize the extra work of a epoch change against the successful operations in the epoch. The claim follows. □

[3] A CAS is successful if it changes the value of the register, and fails otherwise.

B Preventing Data Structure Overflow

We only consider inputs which do not allow the data structure to have more than m items inserted at the same time. Formally, we require that all inputs obey the following property. Recall that an input is a sequence of DoTask and InsertTask operations. A *subsequence* of the input is a *contiguous* substring.

Definition 1. *An input I is* valid *if it has no contiguous subsequence such that the number of* InsertTask *minus the number of* DoTask *operations in the subsequence is greater than $m - 2p$.*

We now prove that no valid input can result in a situation where the data structure has more than m elements.

Lemma 16. *For any valid input I, the task allocation object has at most m available tasks.*

Proof. We proceed by contradiction. Fix a valid input I, and assume that there exists an execution $\mathcal{E}$ on I where the data structure has $m + 1$ tasks available. We show the existence of a contiguous subsequence S of I containing $m - 2p$ more insert operations than remove operations, which contradicts the valid input assumption.

Consider the execution $\mathcal{E}$ up to the first point where $m+1$ tasks are available. At that point, consider the last operation in I that was assigned to some process. Let subsequence S finish with this operation. Now consider the latest point in the execution when some DoTask operation failed (i.e., it observed an empty To-Do tree). All the subsequent Do-Task operations were successful. The process performing that operation gets immediately assigned to the next operation in the input. Let our subsequence S start with this operation. If no DoTask operation failed in the execution, then let S start with the first operation in I.

We have identified two points in the execution, one when the To-Do tree was empty, and another when it contained m tasks. Meanwhile, all except possibly at most p operations from subsequence S have been executed, in addition to at most p other operations that were initially assigned to processes when the last DoTask operation failed. Initially, the data structure has no available tasks. Therefore, subsequence S has to contain at least $m - 2p$ more InsertTask operations than DoTask operations, contradicting the valid input assumption. $\square$

In fact, notice that, if input I contains a contiguous subsequence with $m+1$ more InsertTask operations than DoTask operations, the adversary can easily overflow the To-Do tree. The strategy is to let all the operations located before the beginning of the subsequence finish, and then to execute all operations from the subsequence.