CAFÉ: Scalable Task Pools with Adjustable Fairness and Contention

Dmitry Basin[1], Rui Fan[2], Idit Keidar[1], Ofer Kiselov[1], and Dmitri Perelman[1⋆]

[1] Department of Electrical Engineering,Technion,Haifa,Israel
{sdimbsn@tx,idish@ee,sopeng@t2,dima39@tx}.technion.ac.il
[2] School of Computer Engineering, Nanyang Technological University
fanrui@ntu.edu.sg

Abstract. Task pools have many important applications in distributed and parallel computing. Pools are typically implemented using concurrent queues, which limits their scalability. We introduce *CAFÉ, Contention and Fairness Explorer*, a scalable and wait-free task pool which allows users to control the trade-off between fairness and contention. The main idea behind CAFÉ is to maintain a list of *TreeContainers*, a novel tree-based data structure providing efficient task inserts and retrievals. TreeContainers don't guarantee FIFO ordering on task retrievals. But by varying the size of the trees, CAFÉ can provide any type of pool, from ones using large trees with low contention but less fairness, to ones using small trees with higher contention but also greater fairness.
We demonstrate the scalability of TreeContainer by proving an $O(\log^2 N)$ bound on the step complexity of insert operations when there are N inserts, as compared to an average of $\Omega(N)$ steps in a queue based implementation. We further prove that get operations are wait-free. Evaluations of CAFÉ show that it outperforms the Java SDK implementation of the Michael-Scott queue by a factor of 30, and is over three times faster than other state-of-the-art non-FIFO task pools.

1 Introduction

A *task pool* is a data structure consisting of an unordered collection of objects, a *put* operation to add an object to the collection, and a *get* operation to remove an object[3]. Pools have a number of important applications in multiprocessor computing, such as maintaining the set of pending tasks in a parallel computation. A key challenge in such an application is to ensure the pool does not become a bottleneck when it is concurrently accessed by a large number of threads. Another challenge is to ensure fairness — although strict FIFO ordering is not necessary, we nevertheless want to avoid starvation and limit the number of *overtakings*[4].

In this paper, we present CAFÉ (Contention And Fairness Explorer), an efficient randomized wait-free[5] task pool algorithm. CAFÉ maintains a list of scalable bounded

⋆ This work was partially supported by Hasso Plattner Institute.

[3] We sometimes refer to task pools as producer-consumer pools; producers do puts, and consumers do gets.

[4] One task overtakes another task if it is inserted after the other task, but retrieved before it.

[5] A randomized algorithm is *wait-free* if each thread executing an operation performs a finite number of steps with probability 1.

pools called *TreeContainers*. When one TreeContainer becomes full, a new TreeContainer is appended to the end of the list. Retrievals follow the FIFO order of the TreeContainers, but each TreeContainer can return its tasks in any order. This way, the tree size is a system parameter controlling the trade-off between fairness and contention. Using smaller trees, the system provides better fairness but also has more contention.

A TreeContainer stores jobs in a complete binary tree, in which every node can store one task. Each node keeps presence bits indicating whether its child subtrees contain tasks. This allows get operations to find tasks by walking down the tree from the root, following a trail of presence bits. At the same time, the bits do not change frequently, even when there are a large number of concurrent puts and gets, so they do not cause much contention. We show that TreeContainers are dense: a tree with height h contains at least $2^{(1-\epsilon)h}$ tasks with high probability, for any $\epsilon > 0$. We also show that TreeContainers perform well under contention. When there are N concurrent put operations and an arbitrary number of gets, each put finishes in $O(\log^2 N)$ steps, whp.

CAFÉ combines TreeContainers in a FIFO linked list, to provide the following properties. 1) The number of overtaken tasks in CAFÉ is bounded by the size of a tree. 2) In most workloads, producers and consumers operate on different TreeContainers, which decreases contention and improves performance. 3) Puts are wait-free with probability 1, and gets are deterministically wait-free.

Our algorithm offers some significant advantages over other approaches for task pools. The most common approach to implement pools is using FIFO queues (e.g., Java ThreadPoolExecutor). However, non-blocking queue-based algorithms suffer $\Omega(N)$ contention at the head and tail, while our algorithm has $O(\log^2 N)$ contention for puts, whp. Other queue-based algorithms are blocking, and require puts and gets to wait for each other. In contrast, all operations in our algorithm are wait-free. The recent ED pools in [1] also use trees, but in a different way. Unlike our algorithm, [1] does not provide any upper bounds on step complexity, nor on the number of times a task can be overtaken.

We have implemented CAFÉ in Java, and tested its performance on a 32-core machine[6]. Our results show that CAFÉ is over 30 times faster than a pool based on Java's implementation of the Michael-Scott queue, and over three times faster than a pool using Java's state-of-the-art blocking queue (even though CAFÉ does not block). Also, CAFÉ is over three times faster than ED pools, while providing stronger fairness guarantees.

The remainder of the paper is organized as follows. In Section 2, we describe related work. We present CAFÉ in Section 3, and analyze its theoretical properties in Section 4. We discuss our experimental results in Section 5. Finally, we conclude in Section 6.

2 Related Work

A common approach to implementing concurrent task pools is to use FIFO queues for task management. However, due to their strong ordering guarantees, such implementations are not scalable, suffering from $\Omega(N)$ contention in the worst case. CAFÉ makes

[6] The code is publicly available at `http://code.google.com/p/cafe-pool/`.

the observation that strict FIFO ordering is not necessary for a task pool, and thereby achieves a much more scalable algorithm.

Another approach for reducing contention is using *elimination*, as proposed by Moir et al. [8]. Here, producers and consumers can "eliminate" each other at predefined rendezvous points. This approach best suits workloads in which there are more consumers than producers. Elimination is less useful if the queue remains non-empty most of the time, or when concurrency is low. In contrast, CAFÉ performs well under both high and low concurrency, and regardless of the ratio between producers and consumers[7].

Afek *et al.* [1] also propose a task pool foregoing FIFO ordering for scalability. Their Elimination Diffraction (ED) pools yield significantly better results than FIFO implementations. ED pools use a fixed number of queues along with elimination for reducing contention. However, as we show in Section 5.2, ED pools do not scale well on multi-chip architectures. In addition, unlike CAFÉ , ED pools are not wait-free, and offer no fairness guarantees between puts and gets.

The idea of using concurrent tree-based data structures for reducing contention has appeared in previous works not related to task pools [4, 3]. Unlike these works, we prove formal bounds on the worst case step complexity of our TreeContainer algorithm.

3 CAFÉ: A Task Pool with Adjustable Fairness

In this section, we describe CAFÉ, a wait-free, scalable task pool algorithm, whose fairness can be adjusted arbitrarily by the user. The main idea behind CAFÉ is to keep a linked list of scalable task pools called *TreeContainers*, each with bounded size. The algorithm for a single TreeContainer is given in Section 3.1. Tasks are stored at tree nodes, which can be occupied at most once. When a tree becomes full, a new tree is added to the list. The algorithm for combining TreeContainers in a FIFO list is described in Section 3.2.

3.1 TreeContainer

A TreeContainer consists of a bounded complete binary tree, in which each node can store one task. A node with a task is *occupied*, and otherwise it is *free*. Each node can be occupied at most once, as indicated by an *isDirty* flag. In addition, the node keeps a *presence bit* for each child subtree; the bit is zero when all the nodes in the respective subtree are free. Presence bits allow get operations to find a task in the tree by walking down from the root following a trail of non-zero bits. Since presence bits summarize the occupancy of an entire subtree, they change infrequently even under highly concurrent workloads, which allows our algorithm to achieve low step complexity.

TreeContainer is shown in Algorithm 1. Level i of the tree is implemented using an array tree[i], which allows $O(1)$ access to any node in a level. The root is the only node at level 0. Each node also keeps pointers to its father and children, as well as a bit *side*, indicating whether it is the left or right child of its father.

[7] Due to space limitations, evaluations of CAFÉ on different workloads is deferred to the full paper [2].

Algorithm 1 TreesContainer, a scalable bounded task pool algorithm.

```
 1: TreeNode data structure:
      ▷ ver: version of the metadata
      ▷ p indicates presence of tasks in left/right subtrees
      ▷ ⟨ver, p⟩ is kept by a single AtomicInteger in Java
 2:   [⟨ver, p⟩, ⟨ver, p⟩]: meta
 3:   int: pending
 4:   boolean: isDirty        ▷ true if has been already used
 5:   Data: task
 6:   int: side      ▷ 0 for the left child, 1 for the right child
 7: Tree data structure:
      ▷ tree[i] keeps an array with the nodes of level i
 8:   TreeNode[][]: tree

 9: Function hasTasks(node):
10:   if (node.meta[0].p ∨ node.meta[1].p)
11:     then return 1
12:   else return (node.task ≠ ⊥) ? 1 : 0

13: Function put(task):
14:   node ← findNodeForPut(task)
15:   if (node = ⊥) then return false
16:   updateNodeMetadata(node, 1)
17:   return true

18: Function findNodeForPut(task):
19:   for level = 0, 1, . . . do
20:     trials ← (level < height(root)) ? 1 : k
21:     for i = 1, . . . , trials do
22:       node ← random node in tree[level]
23:       reserved ← putInNode(node,task)
24:       if (reserved ≠ ⊥) return reserved
25:   return ⊥                  ▷ did not succeed in this tree

26: Function putInNode(node, task)
27:   if (node.father ≠ ⊥∧ node.father.task = ⊥)
28:     return putInNode(node.father, task)
29:   if (node.isDirty.CAS(false, true))
30:     node.task ← task; return node
31:   else return ⊥

32: Function get()
33:   while(true):
34:     if (hasTasks(root) = 0) return ⊥
35:     node ← findNodeForGet()
36:     task ← node.task
37:     if (task ≠ ⊥ ∧
          node.task.CAS(task, ⊥) = false) continue
38:     updateNodeMetadata(node, 0)
39:     if (task ≠ ⊥) return task

40: Function findNodeForGet()
41:   node ← root
42:   while(true)
43:     if(node.task≠⊥ ∨
          node.meta[0].p=node.meta[1].p=0)
44:       return node
45:     node ← random child among those with p = 1

46: Function updateNodeMetadata(node, myVal)
47:   trials ← 0;
48:   while(node.father ≠ ⊥)
        ▷ check if my operation has been eliminated
49:     if (myVal ≠ hasTasks(node)) return
50:     fk ← father.meta[node.side].p
51:     if (fk ≠ hasTasks(node) ∨ node.pending > 0)
52:       trials ← trials +1
53:       if (updateFather(node) ≠ success ∧
            trials < 2) continue          ▷ try again
54:     node ← node.father; trials ← 0

55: Function updateFather(node)
56:   node.pending.FetchAndInc()
57:   new ← old ← father.meta[node.side]
58:   new.ver ← new.ver +1; new.p ← hasTasks(node)
59:   success ← father.meta[node.side].CAS(old, new)
60:   node.pending.FetchAndDec()
61:   return success
```

Task Insertion. Tasks are inserted in a tree using the $put()$ operation. First, put finds a free node to insert the task. Then it updates the presence bits of the node's ancestors. Because a tree has bounded size, task insertions can fail if they do not find a free node in the tree. Below, we describe the main steps in a put.

Finding an unoccupied node. Function *findNodeForPut()* finds a free tree node for task insertion. It iterates over the tree levels starting from the root (lines 19–24). At each level, a random node x is chosen, and the algorithm tries to put the task in the *highest* free node on the path from x to the root. This is done using the recursive function *putInNode()* (lines 27–31). Nodes are reserved by CASing the $isDirty$ flag. Having nodes search for a free ancestor increases put's step complexity from $O(h)$ to $O(h^2)$ for a tree with height h (proved in [2]). However, it also creates denser trees with a more balanced node occupation, as we show in Section 4.2 and prove in [2].

If neither x nor its ancestors can be reserved, another random node is checked. At each level except the last one, a single node is checked. The number of nodes checked at the last level is defined by a parameter k, with higher k's resulting in denser trees. We

show in Section 4.2 and prove in [2] that in a tree with height h, at least $2^{\frac{k+2}{k+3}h}$ nodes are occupied before a put operation fails, whp.

Updating ancestors' metadata. After a task is inserted in node x, function *updateNodeMetadata()* updates the presence bits of x's ancestors (lines 48–54). At each node the function checks that the metadata of the father is correct. Contention remains low because in the common case, the presence bits of upper level nodes are not updated when a new task is inserted or removed.

Though the general outline of the algorithm is simple, ensuring linearizability, wait-freedom and low contention require special care, as we describe below.

1. *Ensuring linearizability.* A naïve approach to update x's father's metadata could be to first read the old presence bit of x's father (line 50), then calculate whether x's subtree contains tasks (line 57), and finally CAS a new metadata value if the old value is incorrect (line 59). If the CAS fails, the updater retries. Version numbers are attached to the presence bits in order to avoid ABA problems.

Unfortunately, this simple approach can violate linearizability. Consider nodes x, y and z, where y is the right child of x and z is the right child of y. Node y has a task, so that $x.meta[1].p = 1$. There are two concurrent threads, a consumer t_c that removes the task from y and a producer t_p that inserts a task in z. t_c starts updating the metadata of y's father. It reads the right presence bit at x, which is 1, and decides to update it to 0. We then suspend t_c right before it performs its CAS operation. At this time, t_p starts updating the ancestors of z. It first changes $y.meta[1].p$ from 0 to 1, and then checks the right presence bit at x. Since t_c is paused, $x.meta[1].p$ is still 1, and so t_p decides this value is correct, and terminates. Now t_c resumes, and successfully changes $x.meta[1].p$ to 0. This makes future gets think the tree is empty, so that no get will retrieve t_p's task, violating linearizability.

We solve this problem by letting other threads know about concurrent pending updaters. Whenever a thread t plans to change the metadata of x's father, it increments a *pending* counter at x (line 56); after the update, it decrements the counter (line 60). If a concurrent updater sees $x.pending > 0$, it will update x's father's metadata, regardless of its current value (line 51). This, along with the use of version numbers, will cause the pending thread's CAS to later fail.

2. *Limiting the number of CAS failures.* In the simple algorithm described earlier, an updater thread t that fails to CAS the metadata of x's father will retry the update. This makes t's worst case step complexity linear in the tree size, since every thread that successfully performed an operation in x's subtree can cause t's CAS to fail. However, as we show in the full version of the paper [2], it suffices for t to only try to update x's father's metadata *twice* (line 53). The idea is that if t fails two CASes, then some other thread will have already updated x's father's metadata to the correct value.

3. *Producer/consumer elimination.* We have also adopted the elimination technique used in [8] and [1]. Consider a thread t that inserted a new task at a node, and started updating the node's ancestors. Let x and y be two such ancestors, where y is the father of x. In the function $updateNodeMetadata$, t updated y's metadata (on x's side) to 1 while t was still at x. Thus, if t later arrives at y and sees y's x-side metadata is now 0, it means there has been consumer thread that already removed the task t inserted. In this case, t doesn't need to update any more ancestors, and can terminate early (line 49).

This optimization improves performance in scenarios where multiple producers and consumers are working on the same tree.

We show in Section 4.2 and prove in [2] that put operations in TreeContainer are wait-free. Intuitively, this is because the tree is bounded, and because a thread only tries two updates per node. If the tree has height h, the put performs $O(h^2)$ steps. We show in Section 4.2 that our insertions create a balanced tree, whp. Hence, when the tree contains N tasks, the complexity of a put is $O(\log^2 N)$.

Task Retrieval. The *get()* function in TreeContainer runs in a loop (lines 33–39). If there are no tasks in the tree, as indicated by the presence bits at the root, the function returns $\perp$ (line 34). $get()$ first finds a task at a random node to retrieve from using $findNodeForGet()$, and then updates the metadata of the node's ancestors.

Function *findNodeForGet()* searches for a node to get a task from. When it reaches an unoccupied node, it randomly chooses a nonempty subtree to go down. The randomization reduces contention.

A task T is removed from node x by CASing $x.task$ from T to $\perp$ (line 37). If the CAS succeeds, then the metadata of x's ancestors need to be updated. Otherwise, the algorithm starts a new retrieval attempt. Note that if *findNodeForGet()* finds a node x with $x.task = \perp$, it means that another consumer t_c removed x's task but still hasn't updated x's ancestors. In order to be wait-free, a consumer needs to make sure that it will not arrive to this empty node infinitely many times. Hence, a consumer that arrives at an empty node x updates x's ancestors even though it did not take x's task (line 38). Updating the ancestors is done the same way as after a task insertion, using *updateNodeMetadata()*.

We show in Section 4.2 and prove in [2] that get operations are wait-free. Intuitively, this is because a get thread t_c can only fail to take a task from a previously occupied node x if some other thread took x's task. Then, t_c updates the metadata on the path to the root, so that t_c does not go down the same path again. The bounded number of nodes in a tree then limits the number of unsuccessful get attempts.

3.2 Combining TreeContainers in a FIFO List

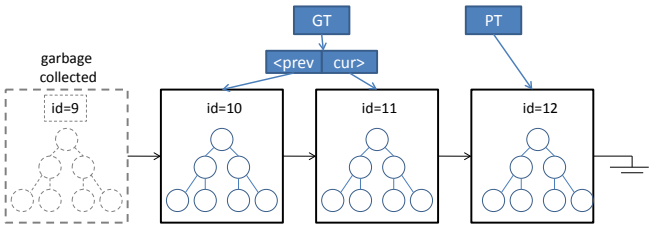


Fig. 1. CAFÉ keeps a linked list of scalable task trees. The tree height defines the fairness of the protocol.

As stated earlier, CAFÉ maintains a linked list of TreeContainers, adding new trees as old ones become full (see Figure 1). Tasks are returned in FIFO order, up to the tree they are inserted into. This guarantees that the maximum number of overtakers in CAFÉ is bounded by the tree size. Therefore, the tree size is a parameter that determines the trade-off between fairness and contention. Using bigger trees, CAFÉ performs more like a TreeContainer, and so has low contention but less fairness. Using smaller trees, CAFÉ performs more like a FIFO list, so there is higher contention but greater fairness.

Basic approach. A simple way to manage a linked list of trees is to keep one pointer (PT) for producers, which references the tree for puts, and another (GT) for consumers, referencing the tree for gets. Whenever the current insertion tree becomes full, PT is moved forward. Whenever no tasks are left in the retrieval tree, GT is moved forward. Old trees are garbage collected automatically in managed memory systems as they become unreachable.

This straightforward approach, however, violates correctness, as we now demonstrate. Consider the following scenario. t_p inserts a task in tree T and pauses before changing the metadata of T's root. Consumers assume that T is empty and increment GT to continue to later trees. When t_p finally resumes, we have $GT > PT$, and no consumer will ever retrieve t_p's task.

One way to solve this problem is to reinsert the task in a later tree whenever t_p notices its task may be lost. However, this approach might lead to livelocks, in which producers constantly chase consumers, never finishing their operations. Another method is to maintain a non-zero indicator on each tree (e.g., using SNZI [3]) indicating whether there are concurrent producers working on the tree. But this approach incurs high overhead, for managing both indicators and lists of "pending and active" trees. Our solution is instead based on the idea of moving the consumer pointer GT backwards when a task is added in an old tree.

Managing the list of trees. The pseudo-code for the list of trees pool is shown in Algorithm 2. A put operation tries to insert the task into the tree pointed to by PT (call this tree T). If the insert fails, the algorithm moves to the next tree in the list by incrementing PT (lines 16–17). New trees are created and appended to the end of the list as needed. For reasons we explain later, the pointer for consumers GT actually points to two consecutive trees, $GT.cur$ and $GT.prev$. When an insert succeeds, the producer checks that its task will be retrievable in the future. To this end, it checks that $GT.cur$ does not point to a tree that succeeds T in the linked list (line 13). If it does, the GT pair is moved backwards to $\langle \perp, T \rangle$ in the function $moveGTBack$.

In $moveGTBack$, a producer repeatedly tries to CAS GT to T until a CAS succeeds, or it reads $GT.cur \leq T$. As we want producers to be wait-free, we need to ensure this loop eventually terminates. Thus, we do not allow the GT pointers to move forward while there are pending producers that want to move GT backwards. We increment a counter $oldProducers$ at the start of $moveGTBack$, and decrement it at the end. If a consumer does not find a task in the GT trees, but sees $oldProducers > 0$, it advances to a later tree, but does not increment GT (line 44).

A consumer tries to retrieve a task from the trees pointed to by $GT.prev$ and $GT.cur$ (lines 36–37). If both trees are empty, and if PT points to a later tree than

Algorithm 2 CAFÉ algorithm for adjustable fairness and contention.

```
1:  Data structures:
2:      Node:
3:          int: id
4:          ScalableTree: tree
            Node: next

5:  Global variables:
6:      Node: PT                          ▷ tree for producers
7:      ⟨prev, cur⟩: GT                   ▷ tree for consumers
8:      int: oldProducers                 ▷ for moving GT backwards

9:  Function put(task)
10:     while(true)
11:         latest ← PT
12:         if (latest.tree.put(task) = true) then
13:             if (GT.cur.id > latest.id)
                        moveGTBack(latest)
14:             return
15:         else
16:             if(latest.next = ⊥) insertNewTree()
17:             PT.CAS(latest, latest.next)

18: insertNewTree()
19:     newNode ← Node()
20:     cur ← PT ▷ go to the end of the list
21:     for(; cur.next ≠ ⊥; cur ← cur.next);
22:     newNode.id ← cur.id +1
23:     cur.next.CAS(⊥, newNode)  ▷ return even if CAS
                                    fails

24: Function moveGTBack(Node: prodTree)
25:     oldProducers.FetchAndInc()
26:     while(true)
27:         gtVal ← GT
28:         if (gtVal.cur.id ≤ prodTree.id) break
29:         newGT ← ⟨⊥, prodTree⟩
30:         if (GT.CAS(gtVal, newGT) = true) break
31:     oldProducers.FetchAndDec()

32: Function get()
33:     ptVal ← PT
34:     gtVal ← GT
35:     while(true)
36:         task ← gtVal.prev.getTask()
            if (task ≠ ⊥) return task
37:         task ← gtVal.cur.getTask()
            if (task ≠ ⊥) return task
            ▷ could not find a task in the tree
38:         if (ptVal.id ≤ gtVal.cur.id) return ⊥
39:         if (oldProducers = 0) then
40:             newGT ← ⟨gtVal.cur, gtVal.cur.next⟩
41:             GT.CAS(gtVal, newGT)
42:             gtVal ← GT
43:         else
44:             gtVal ← ⟨gtVal.cur, gtVal.cur.next⟩
```

$GT.cur$, then GT is updated to $\langle GT.cur, GT.cur.next\rangle$. This update is performed by first creating a pair with the new tuple values (line 40), and then CASing GT from the old pair to the new one (line 41). Note that the ABA problem does not occur during the CAS, because every newly created pair is a new object whose address is different from the addresses of any old pairs, which are not deallocated throughout the function's execution.

Finally, we explain the reason for using two consumer pointers, $GT.cur$ and $GT.prev$. Suppose GT only pointed to one tree, and consider the following situation. GT and PT both point to a tree T. Producer t_p inserts a new task in T and pauses. Meanwhile, other producers insert new tasks, append new trees and move PT. Suppose a consumer t_c comes to retrieve a task, does not find any tasks in T, and pauses right before changing GT to $T.next$. When t_p resumes, it inserts its task to T, checks that GT is still pointing to T and terminates. When t_c resumes, it changes GT to $T.next$. Now, t_p's task is lost. As we show in the next section, keeping two pointers allows us to solve this problem in a simple and efficient way.

In the next section, we show that both put and get operations in CAFÉ terminate within a finite number of steps with probability 1. Thus, CAFÉ is wait-free.

4 CAFÉ's Properties

In this section, we present the correctness and performance properties of CAFÉ. Due to space limitations, we only state the main results and describe the ideas behind them, deferring the full proofs to the full paper [2]. For all the results we assume that an

adversary controls thread scheduling but cannot influence the randomness threads use. We let h denote the height of a TreeContainer, and k denote the number of insertion attempts in the last layer of TreeContainer (line 20 in Algorithm 1).

4.1 Safety Properties

In this section we present safety proof outline. We start by showing that CAFÉ implements a linearizable job pool. Intuitively, if the job pool is nonempty, then a get must be able to find a job. We prove a theorem showing that after any put operation finishes, no subsequent get operation will return $\perp$, until the put's task has been returned.

Theorem 1. *Suppose a get operation g in CAFÉ returns $\perp$ at a time t. Then for every put operation p that completed before the start of g, p's task was removed by some get operation before t.*

The Theorem 1 proof consists of two parts. First, we prove that each TreeContainer CAFÉ uses is itself a linearizable job pool. We formalize this in Lemma 1.

Lemma 1. *TreeContainer implements a linearizable producer-consumer pool.*

Second, we prove that after a put inserts a task in some TreeContainer, subsequent get operations will not skip this TreeContainer when looking for a job. We formalize this in Lemma 2.

Lemma 2. *Let p be a completed put operation that inserted a task in TreeContainer T. Suppose at some time τ, p's task has not been removed. Then $GT.cur.id \leq T.id + 1$ at τ.*

The key to proving Lemma 1 is Lemma 3.

Lemma 3. *Consider any TreeContainer T, and let p be a completed put operation that inserted a task in node $x_0 \in T$. Suppose that by some time τ, no get operation has removed the task from x_0,* i.e. *line 37 with $node = x_0$ has not occurred (Algorithm 1). Then for every node x on the path from x_0 to the root of T, $hasTasks(x) = 1$ at τ.*

The lemma proves that after a put operation has inserted a task in some node of a TreeContainer, $hasTasks(x) = 1$ for every node x on the path from that node to the root of the TreeContainer, until the node's task is removed. We say that the nodes on the path are *marked*. Get operations follow a path of marked nodes, and so will always find a job as long they have not all been removed. We briefly describe the proof of Lemma 3. Let x and y be two nodes a put operation p passes through during $updateNodeMetadata$, where y is the father of x. The invariant we maintain is that the value of $hasTasks(x)$ has been fixed to 1 by the time p starts updating y's metadata. Since p tries to set y's metadata to $hasTasks(x)$, then $hasTasks(y)$ will also be fixed to 1 after p finishes processing y. Thus, all the $hasTasks$ values on the path from p's insertion node to the root will be fixed to 1 inductively.

Next, we briefly describe the proof of Lemma 2. After a put operation has inserted a task in a tree T, it does *moveGTBack* to ensure the value of GT is at most T. There are two ways the put checks this condition. Either it successfully CASed the value $\langle \perp, T \rangle$

into GT, or it read that $GT.cur$ is at most T. Because the CASes on GT can be linearized, we can show in the first case that later gets see T (or a smaller value) when they read GT. In the second case, we need to be careful that while the put is checking $GT.cur$ is at most T, there may be a paused get operation, which then increases GT as soon as the put's check finishes. However, even if this happens, $GT.cur$ only moves forward by 1. Since a get operation checks both $GT.cur$ and its preceding tree $GT.prev$, the get will still see the tree that the put inserted into.

The last correctness property we show is that gets return jobs in FIFO order, up to the TreeContainer they were inserted into. This follows simply because jobs are inserted and removed based on the linked list order of the TreeContainers.

4.2 Performance Properties

We first show that our trees are dense: by choosing an appropriate k we can guarantee that a tree with height h is populated with at least $2^{(1-\epsilon)h}$ tasks for an arbitrary $0 < \epsilon < 1$, with high probability. In the full paper [2], we also show that this density is higher than that achieved by a simple random walk based insertion. More formally, we prove the following lemma.

Lemma 4. *In a TreeContainer of height h, if a put operation fails, then the tree contains at least $2^{\frac{k+2}{k+3} \cdot h}$ tasks with probability at least $1 - \frac{1}{2^{(3-\frac{7}{k+3})h+k+1}}$.*

In addition, we prove that TreeContainer has a bound on put operation step complexity:

Lemma 5. *Every* put() *operation of TreeContainer makes at most $O(h^2)$ steps.*

We further demonstrate that TreeContainer performs well under contention. For N concurrent put operations and an arbitrary number of get operations, each put finishes in $O(\log^2 N)$ steps, whp:

Lemma 6. *Consider a TreeContainer after N successful* put *operations. Then each of these operations has taken $O(\log^2 N)$ steps with probability at least $1 - \frac{1}{2(N+1)^{\frac{4}{3}}}$.*

We next intuitively demonstrate the wait-freedom of CAFÉ. We first show that put operations are wait-free with probability 1, and then argue that get operations are deterministically wait-free.

A put operation traverses the linked list of TreeContainers until it successfully inserts a task in one of them; new TreeContainers are appended if the insertions keep failing. Intuitively, it might seem that this traversal could go on forever. For example, a slow thread t_p could repeatedly try to insert a task in some tree, then pause until all other producers proceed to a new tree, fail its current insert, and have to retry in a new tree. Fortunately, this situation does not happen. Due to the randomness in the algorithm, other threads are likely to have left unoccupied nodes in t_p's tree, which t_p can acquire once it resumes. We formalize this intuition in the following lemma.

Lemma 7. *If P producer threads and any number of consumer threads use CAFÉ, then any TreeContainer's* put *operation succeeds with probability at least $(1 - \frac{1}{2^h})^{k(P-1)} \cdot [1 - (1 - \frac{1}{2^h})^k]$.*

Using Lemma 7, we prove the following. Note that CAFÉ using TreeContainers of height 0 is equivalent to a linked list.

Lemma 8. *If the height of TreeContainer is greater than zero, then CAFÉ's* put *operations are wait-free with probability* 1.

In order to show CAFÉ 's get operations are wait-free, we need to show that a consumer does not need to traverse an unbounded number of trees when looking for a task. This is true because each get operation keeps a pointer to the latest TreeContainer when it starts (line 33 in Algorithm 2), and subsequently only checks trees that had tasks before it started. In a linearizable execution, the get is allowed to return $\perp$ when all these trees are empty (in line 38), as all their tasks will have been taken by other gets concurrent with or preceding the current get. We conclude with the following lemma.

Lemma 9. *Every get operation of CAFÉ terminates in a finite number of steps.*

5 Evaluation

In this section we evaluate the performance of Java implementation of CAFÉ. Due to space limitations, we only describe the highlights of our evaluation. More comprehensive experimental results may be found in the full paper [2].

5.1 Experiment Setup

We compare the following task pool implementations:

- **CAFÉ-h** – CAFÉ with height h for each tree. Unless stated otherwise, we use $h = 12$.
- **CLQ** – The standard Java 6 implementation of a (FIFO) non-blocking queue by Michael and Scott [7] (class `java.util.concurrent.ConcurrentLinkedQueue`, which is considered to be one of the most efficient non-blocking algorithms in the literature [5, 6].
- **LBQ** – The standard Java 6 implementation of a (FIFO) blocking queue that uses a global reader-writer lock (class `java.util.concurrent.LinkedBlockingQueue`).
- **ED** – The original elimination-diffraction tree implementation [1] (downloaded from the web page of the project), in its default configuration. Tasks are inserted into a diffraction tree with FIFO queues attached to each leaf. The queues are implemented using Java LinkedBlockingQueues. Every tree node contains an elimination array where producers can pass tasks directly to consumers. Changing the tree depth, pool size and spinning behavior did not have a significant effect on the pool's performance. Note that ED trees, like CAFÉ , do not enforce FIFO ordering.

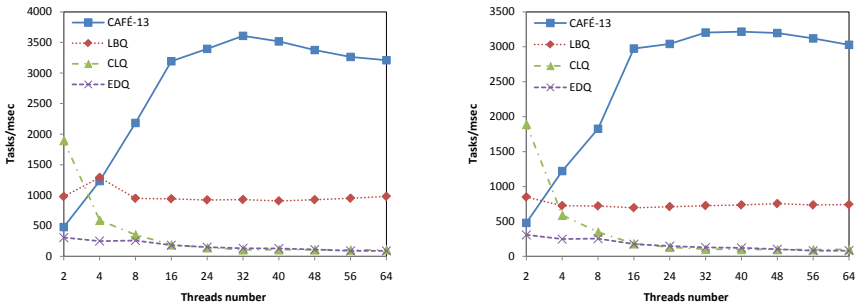
We use a synthetic benchmark for the performance evaluation, in which producer threads work in loops inserting dummy items, and consumer threads work in loops retrieving dummy items.

Unless stated otherwise, tests are run on a dedicated shared memory NUMA server with 8 Quad Core AMD 2.3GHz processors and 16GB of memory attached to each

processor. JVM is run with the AggressiveHeap flag on. We run up to 64 threads on the 32 cores. The influence of garbage collection was negligible for all algorithm[8].

We analyze system performance in Section 5.2 and study the influence of tree heights in Section 5.3.

5.2 System Throughput



(a) Task insertion rate.

(b) Task removal rate.

Fig. 2. Task insertion and retrieval rates (equal numbers of producers and consumers). The throughput of CAFÉ-13 increases up to 32 threads (the number of hardware threads in the system). In this configuration it is $\times 30$ faster than the Michael-Scott ConcurrentLinkedQueue and over three times higher than all other implementations, including the ones not providing FIFO. CAFÉ continues demonstrating high throughput even when the number of threads increases up to 64.

In Figure 2 we show the average insertion and retrieval rates in a system with an equal number of producers and consumers. Both graphs demonstrate the same behavior. The throughput of CAFÉ increases up to 32 threads, the number of hardware threads in our architecture. At this point, the throughput of CAFÉ is $\times 30$ higher than the Michael-Scott queue or the ED pool. It is also over three times higher than the blocking queue. When the number of working threads exceeds the number of hardware threads in the system, the throughput of CAFÉ decreases moderately, but still outperforms the other algorithms.

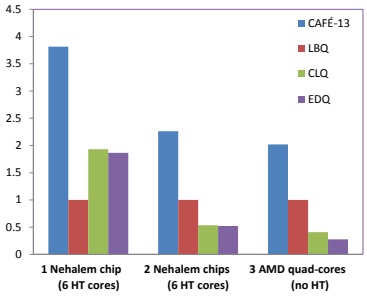


Fig. 3. Throughput on different hardware architectures, normalized by the throughput of LBQ. There are 6 producer threads and 6 consumer threads.

As we can see in Figure 2, the results of both the Michael-Scott concurrent queue and ED pools are lower than those of other algorithms. This differs from the results demonstrated by Afek *et al.* [1], where ED pools were shown to clearly outperform

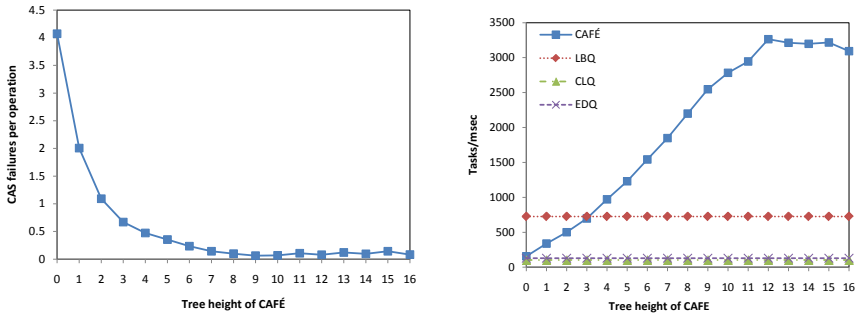
[8] This was checked using the verbose:gc flag in JVM.

standard Java queues. This discrepancy seems to follow from differences in the hardware architectures used in our experiments. Afek *et al.* use a Sun UltraSPARC T2 machine with 2 processors of 64 hardware threads each, while in our system there are 8 quad-cores. The difference in architecture is significant due to the *non-uniform* memory access time in multi-processor systems: accessing a memory location from multiple processors is significantly slower than accessing it from multiple hardware threads on the same chip, which usually share a last-level cache. We now show how the non-uniformity of memory accesses influences performance.

Figure 3 demonstrates the throughput of the algorithms in three different configurations: a single Nehalem chip with 6 hyper-thread cores, two Nehalem chips with 6 hyper-thread cores and three AMD quad-cores with no hyper-threading. The algorithms are run with 6 producers and 6 consumers (corresponding to the number of hardware threads available in a single Nehalem chip); the throughput is normalized by the throughput of the Java LinkedBlockingQueue.

We observe that, consistent with the findings of Afek *et al.*, both ED pools and MS non-blocking queue perform twice as well as Java's linked blocking queue when running on a single chip. However, their performances decrease significantly in systems with two or more chips, when memory sharing becomes more expensive. We point out that CAFÉ continues to outperform all the other algorithms even in the single-chip case, which is the best setting for ED pools and the MS queue. Nevertheless, it is worth mentioning that in [1], ED pools achieved the best results when run on many threads (up to 64) on the same core. We were unable to reproduce these results as we do not have a machine with more than 12 HW threads per chip.

5.3 Choosing the Tree Height



(a) CAS failures per operation as a function of tree height. (b) CAFÉ throughput as a function of tree height.

Fig. 4. CAS failures and system throughput as a function of CAFÉ's tree height for 16 producers and 16 consumers. Small trees induce high contention because of linked list manipulations and reduced tree randomization. Excessively large trees induce contention among producers and consumers operating in the same tree.

In Figure 4 we demonstrate CAFÉ's performance for 16 producers and 16 consumers as a function of tree height. Figure 4(a) shows the average number of CAS

failures per insertion / removal operation. For height $= 0$, CAFÉ is equivalent to the Michael-Scott concurrent queue, and there are 4 CAS failures per operation. The rate of CAS failures drops quickly for larger trees, becoming less than 0.1 for CAFÉ-8.

The statistics of CAS failures match the throughput graph shown in Figure 4(b). Increasing the tree height improves throughput up to a certain point (12 in our workload), but beyond this performance plateaus. This is because for intermediate tree sizes, producers and consumers usually find themselves in different trees (the latter lagging behind the former), while for heights larger than 13, most of the threads operate in the same tree, which increases contention and decreases performance.

6 Conclusions

We presented CAFÉ, an efficient wait-free task pool with adjustable fairness and contention. CAFÉ uses a scalable TreeContainer building block, which greatly improves on the performance of queue-based alternatives and provides polylogarithmic step complexity for its put operations. Our evaluations show that CAFÉ significantly outperforms both FIFO and non-FIFO task pool algorithms in multi-chip architectures. As we've seen, existing task pools make different trade-offs between fairness and contention. We believe an interesting theoretical question is whether this trade-off is inherent.

References

1. Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Euro-Par 2010 - Parallel Processing*, pages 151–162. 2010.
2. D. Basin, R. Fan, I. Keidar, O. Kiselov, and D. Perelman. Scalable producer-consumer task pools with adjustable fairness and contention. Technical report, Technion, CCIT 790, 2011.
3. F. Ellen, Y. Lev, V. Luchangco, and M. Moir. Snzi: scalable nonzero indicators. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 13–22, 2007.
4. J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, ASPLOS-III, pages 64–75, 1989.
5. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
6. E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free fifo queues. *Distributed Computing*, 20:323–341, 2008.
7. M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, 1996.
8. M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 253–262, 2005.