Allocate-On-Use Space Complexity of Shared-Memory Algorithms

James Aspnes[1]
Yale University Department of Computer Science, New Haven, CT, USA

Bernhard Haeupler
Carnegie Mellon School of Computer Science, Pittsburgh, PA, USA

Alexander Tong[2]
Yale University Department of Computer Science, New Haven, CT, USA

Philipp Woelfel
University of Calgary, Department of Computer Science, Calgary, AB, Canada

Abstract

Many fundamental problems in shared-memory distributed computing, including mutual exclusion [8], consensus [18], and implementations of many sequential objects [14], are known to require linear space in the worst case. However, these lower bounds all work by constructing particular executions for any given algorithm that may be both very long and very improbable. The significance of these bounds is justified by an assumption that any space that is used in some execution must be allocated for all executions. This assumption is not consistent with the storage allocation mechanisms of actual practical systems.

We consider the consequences of adopting a per-execution approach to space complexity, where an object only counts toward the space complexity of an execution if it is used in that execution. This allows us to show that many known randomized algorithms for fundamental problems in shared-memory distributed computing have expected space complexity much lower than the worst-case lower bounds, and that many algorithms that are adaptive in time complexity can also be made adaptive in space complexity.

For the specific problem of mutual exclusion, we develop a new algorithm that illustrates an apparent trade-off between low expected space complexity and low expected RMR complexity. Whether this trade-off is necessary is an open problem.

For some applications, it may be helpful to pay only for objects that are updated, as opposed to those that are merely read. We give a data structure that requires no space to represent objects that are not updated at the cost of a small overhead on those that are.

2012 ACM Subject Classification Theory of computation → Distributed computing models, Computing methodologies → Shared memory algorithms, Software and its engineering → Mutual exclusion

Keywords and phrases Space complexity, memory allocation, mutual exclusion

Digital Object Identifier 10.4230/LIPIcs.DISC.2018.8

[1] Supported in part by NSF grants CCF-1637385 and CCF-1650596.

[2] Supported by NSF grant CCF-1650596.

32nd International Symposium on Distributed Computing (DISC 2018).
Editors: Ulrich Schmid and Josef Widder; Article No. 8; pp. 8:1–8:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The space complexity of shared-memory distributed data structures and protocols, measured in terms of the number of distinct objects needed to implement them, is typically linear in the number of processes. On the upper bound side, this follows from the ability to implement most algorithms using a single output register for each process (which might hold very large values). On the lower bound side, linear lower bounds have long been known for fundamental problems like mutual exclusion [8] and implementing many common shared-memory objects [14]; and have been shown more recently for consensus [10, 18].

Linear bounds are not terrible, but they do limit the scalability of concurrent data structures for very large numbers of processes. The structure of the known lower bound proofs suggest that executions requiring linear space may be rare: known bounds on mutual exclusion and perturbable objects may construct exponentially long executions, while the bounds on consensus depend on constructing very specific executions that are avoidable if the processes can use randomization.

We propose considering per-execution bounds on the space complexity of a shared-memory protocol, where the protocol is charged only for those objects that it actually uses during the execution. This allows for expected space-complexity bounds and high-probability space complexity bounds, which would be meaningless if an algorithm is charged for all objects, used or not.

We define this measure formally in Section 2. We believe that our measure gives a more refined description of the practical space complexity of many shared-memory algorithms, and observe in our analysis of previously known algorithms in Section 3 that our measure formalizes notions of allocate-on-use space complexity that have already been informally considered by other researchers.

Charging only for objects used has strong practical justifications:

- In a system that provides storage allocation as part of its memory management, it may be that unused registers or pages have no actual cost to the system. Alternatively, it may be possible to construct high-level storage allocation mechanisms even in an adversarial setting that allow multiple protocols with dynamic space needs to share a large fixed block of memory.
- Given an algorithm with low expected space complexity – or better yet, with high-probability guarantees of low space complexity – we may be able to run it in fixed space at the cost of accepting a small chance that the algorithm fails by attempting to exceed its space bound. Thus randomized space complexity can be a tool for trading off space for probability of failure.

To show the applicability of our measure, we also include several positive results: In Section 3, we demonstrate that many known algorithms for fundamental shared-memory algorithms either have, or can be made to have with small tweaks, low space complexity in most executions. In Section 4, we describe a new randomized algorithm for mutual exclusion that achieves $O(\log n)$ space complexity with high probability for polynomially many invocations.

In Section 5, we consider an alternative measure that charges only objects that are updated and not those that are only read. We show that this is equivalent up to logarithmic factors to the allocate-on-use measure.

Finally, we discuss open problems in Section 6.

1.1 Model

We consider a standard asynchronous shared-memory model in which a collection of n **processes** communicate by performing **operations** on shared-memory **objects**. Concurrency is modeled by interleaving operations; each operation takes place atomically and is a **step** of the process carrying it out. For convenience, we assume that the identity of an operation includes the identity both of the process carrying out the operation and of the object to which it is applied. An **execution** is a sequence of operations.

Scheduling is controlled by an adversary. If the processes are randomized, then each process has access to local coins that may or may not be visible to the adversary. An **adaptive adversary** may observe the internal states of the processes, including the results of local coin-flips, but cannot predict the outcome of future coin-flips. An **oblivious adversary** simply provides in advance a list of which process carries out an operation at each step, without being able to react to the choices made by the processes. We may also consider adversaries with powers intermediate between these two extremes. In each case, the interaction between the processes and the adversary gives a probability distribution over executions. But rather than make this probability distribution explicit, we will usually just generalize the notion of an execution to a random variable H that maps to each possible execution with a probability determined by the distribution.

1.1.1 Time complexity

The **individual step complexity** of an algorithm executed by a single process is the number of steps carried out by that process before it finishes. The **total step complexity** is the total number of steps over all processes. For mutual exclusion algorithms, we may consider the **remote memory reference** (**RMR**) complexity, in which read operations on a register are not counted if (a) the register has not changed since the last read by the same process (in the **distributed shared memory model** or (b) no operation has been applied to the registers since the last read by the same process (in the **cache-coherent model**).

2 Space complexity

The traditional measure of space complexity is **worst-case space complexity**, the number of distinct objects used by the protocol across all executions. We consider instead the space complexity of individual executions.

▶ **Definition 1.** The **space complexity** of an execution H of a shared-memory system is the number of distinct objects O such that H includes at least one operation on O.

For randomized algorithms, this allows us to talk about **expected space complexity** – the expected value of the space complexity of the execution resulting from the random choices of the processes – and **high-probability** bounds on space complexity – where the bound applies to the space complexity of all but a polynomially-small fraction of executions.

For adaptive algorithms, this allows the space complexity of an execution to depend on the number of participating processes.

3 Examples of allocate-on-use space complexity

In this section, we analyze the space complexity of several recent algorithms from the literature. These include the current best known algorithms (in terms of expected individual step complexity) for implementing test-and-set [11] and consensus [3] from atomic registers, assuming an oblivious adversary. We also include some related algorithms to demonstrate how charging only for objects used can illuminate trade-offs that might not otherwise be visible.

▶ **Theorem 2.**

1. *Let H be an execution of the RatRace algorithm for adaptive test-and-set of Alistarh et al. [2], with k participants. Then the space complexity of H is $\Theta(k)$ with high probability.*
2. *Let H be an execution of the randomized test-and-set of Alistarh and Aspnes [1]. Then the space complexity of H is $\Theta(\log\log n)$ with high probability.*
3. *Let H be an execution of the randomized test-and-set of Giakkoupis and Woelfel [11]. Then the space complexity of H is $\Theta(\log n)$ with high probability.*
4. *Let H be an execution of the $\Theta(\log\log n)$ expected time m-valued randomized consensus protocol of Aspnes [3]. Then the space complexity of H is $\Theta\left(\log\log n \cdot \frac{\log m}{\log\log m}\right)$ in expectation.*

Proof.

1. The RatRace algorithm works by having each processes randomly select a path through a binary tree until it manages to acquire a node using a splitter [16], then fight its way back to the root by winning a 3-process consensus object at each node. Both the splitter and consensus object associated with each node require a constant number of registers to implement, so the space complexity is determined by the number of nodes in the subtree traversed by processes. An analysis of a similar algorithm for adaptive collect [6] is used to show that the size of the tree is $\Theta(k)$ with high probability, so $\Theta(k)$ of the $O(n^3)$ registers pre-allocated in the RatRace algorithm are used. This implies that the algorithm uses $\Theta(k)$ space with high probability.

   Because our model does not require pre-allocating a specific bounded address space, RatRace can be modified to use an unbounded number of possible processes and still get the claimed bounds as a function of k.
2. The Alistarh-Aspnes TAS runs the processes through a sequence of $\Theta(\log\log n)$ **sifter** objects, each implemented using a one-bit atomic register. The authors show that with high probability, a constant number of processes remain at the end of this sequence, which then enter a RatRace TAS object. The sifter array uses $\Theta(\log\log n)$ space in all executions. From the previous argument, the RatRace object uses $O(1)$ space with high probability.
3. The Giakkoupis-Woelfel TAS also uses a sequence of sifter objects; these reduce the number of remaining processes to $O(1)$ in only $\Theta(\log^* n)$ rounds, but the cost is an increase in the space required for each object to $O(\log n)$. However, after the first sifter the number of remaining processes drops to $O(\log n)$ with high probability, so subsequent sifter objects can be implemented in $O(\log\log n)$ space. This makes the space required dominated by the initial sifter object, giving the claimed bound.
4. The Aspnes consensus algorithm uses a sequence of rounds, where each round uses a structure based on the Alistarh-Aspnes sifter to reduce the number of distinct identities followed by an adopt-commit object to detect agreement. This produces agreement in $\Theta(\log\log n)$ rounds on average.

   Using the adopt-commit of Aspnes and Ellen [4], we get $\Theta(1)$ space for each round for the sifter plus $\Theta\left(\frac{\log m}{\log\log m}\right)$ for the adopt-commit object. Multiplying by $\Theta(\log\log n)$ expected rounds gives the claimed bound. ◀

Curiously, all of the variation in space usage for the test-and-set algorithms analyzed above can be attributed to RatRace, either by itself or as a backup for a faster algorithm for winnowing the processes down to a constant number. Using a worst-case measure of space complexity hides the cost of these winnowing steps behind the polynomial worst-case space complexity of RatRace. Using our measure instead exposes an intriguing trade-off between time and space complexity, where the Alistarh-Aspnes algorithm obtains $O(\log\log n)$ space complexity at the cost of $\Theta(\log\log n)$ individual step complexity, while the Giakkoupis-Woelfel algorithm pays $O(\log n)$ space complexity but achieves a much better $\Theta(\log^* n)$ individual step complexity. Whether this trade-off is necessary is an open problem.

4 Monte Carlo Mutual Exclusion

In this section, we present a Monte Carlo mutual exclusion algorithm, which uses only $O(\log n)$ registers, and against a weak adaptive adversary satisfies mutual exclusion with high probability for polynomially many passages through the critical section. This can be used directly, or can be combined with Lamport's fast mutual exclusion algorithm [15] to give an algorithm that uses $O(\log n)$ space initially, then backs off to a traditional $O(n)$ space algorithm when the Monte Carlo algorithm fails. The combined algorithm thus guarantees mutual exclusion in all executions while using only $O(\log n)$ space with high probability for polynomially many passages through the critical section.

A mutual exclusion algorithm provides two methods, `lock()` and `unlock()`. Each process repeatedly calls `lock()` followed by `unlock()`. When a process's `lock()` call terminates, it is in the **critical section** (CS). The algorithm satisfies **mutual exclusion**, if for any execution, no processes are in the critical section at the same time. An infinite execution is **fair**, if each process that is in the CS or has a pending `lock()` or `unlock()` call either takes infinitely many steps or enters the remainder section (which happens when it is not in the CS and has no `lock()` or `unlock()` call pending). A mutual exclusion algorithm is **deadlock-free**, if in any infinite fair execution, each `lock()` and `unlock()` call terminates. If it is randomized, and in an infinite fair execution each `lock()` and `unlock()` call terminates with probability 1, then we call it **randomized deadlock-free**.

Burns and Lynch [8] proved that any deterministic deadlock-free mutual exclusion algorithm implemented from registers, requires at least n of them. For fewer than n registers, the proof constructs exponentially long executions such that at the end two processes end up in the CS. But there are no mutual exclusion algorithms known that use $o(n)$ registers and do not fail provided that only polynomially many `lock()` calls are made. Here we present a randomized algorithm that has this property with high probability, i.e., it uses only $O(\log n)$ registers, and in an execution with polynomially many `lock()` calls mutual exclusion is satisfied with high probability.

Our algorithm works for a weak adaptive adversary, which cannot intervene between a process's coin flip and its next shared step. I.e., it schedules a process based on the entire system state, and then that process flips its next coin, and immediately performs its following shared memory step.

The time efficiency of mutual exclusion algorithms is usually measured in terms of remote memory references (RMR) complexity. Here we consider the standard cache-coherent (CC) model. Each processor keeps local copies of shared variables in its cache; the consistency of copies in different caches is maintained by a coherence protocol. An RMR occurs whenever a process writes a register (which invalidates all valid cache copies of that register), and when a process reads a register of which it has no valid cache copy. The RMR complexity of a mutual exclusion algorithm is the maximum number of RMRs any `lock()` and `unlock()` method incurs. The best deterministic mutual exclusion algorithms use $O(n)$ registers and have an RMR complexity of $O(\log n)$ [17], which is tight [5]. Randomized Las Vegas algorithms can beat the deterministic lower bound (e.g. [7, 13, 12]), but they all use at least a linear or even super-linear number of registers and stronger compare-and-swap primitives.

Our algorithm has an expected *amortized* RMR complexity of $O(n)$: In any execution with L `lock()` calls, the total expected number of RMRs is $O(n \cdot L)$.

4.1 The algorithm

Pseudocode for our Monte Carlo mutual exclusion algorithm is given in Figure 1.

The idea of the algorithm is that to reach the critical section, a process must climb a slippery ladder whose rungs are a set of $\Gamma = O(\log n)$ Boolean registers $S_0, \ldots, S_{\Gamma-1}$. Each of these registers is initially 0.

To climb the ladder, a process executing a `lock()` call attempts to acquire each rung by flipping a coin. With probability 1/2, it writes a 1 to the register and continues to the next. With probability 1/2, it reads the register instead. If the process reads a 0, it tries again; if a 1, it falls back to the bottom of the ladder. The first process to write a 1 will always rise, preventing deadlock. Roughly half of the remaining processes that reach each rung will fall, leaving only a single process with high probability after $O(\log n)$ rungs. For processes that fall, the number of steps they take in their ascent has a geometric distribution, so each such process takes $O(1)$ steps per attempt.

At the bottom of the ladder, a process spins on an auxiliary register A, that is modified only by processes executing `unlock()` calls. This ensures that the expected amortized RMR complexity of each passage through the critical section is $O(n)$, as each call to `unlock()` releases at most n processes, each of which takes $O(1)$ steps before spinning on A again.

To avoid ABAs, register A stores a sequence number that increases with each write. This means that for infinitely many `lock()` calls, the values stored in A are unbounded. But if each process calls `lock()` at most polynomially many times (after which no guarantee for the mutual exclusion property can be made anyway), then $O(\log n)$ bits suffice for A.

▶ **Theorem 3.** *There is a randomized exclusion algorithm implemented from $O(\log n)$ bounded shared registers with expected amortized RMR complexity $O(n)$, such that for a polynomial number of* `lock()` *calls, the algorithm is randomized deadlock-free, and satisfies mutual exclusion with high probability.*

4.2 Proof of Theorem 3

The proof is divided into three parts. Section 4.2.1 shows mutual exclusion holds for polynomially many passages through the critical section with high probability. Section 4.2.2 shows deadlock-freedom. Section 4.2.3 gives the bound on RMR complexity.

4.2.1 Mutual exclusion

We consider a random execution of the algorithm, and let C_t denote the configuration reached at point t, and L_t denote the number of completed `lock()` calls at point t.

The idea of this part of the proof is that we define a potential function $\Phi(C_t)$ that becomes exponentially large if more than one process enters the critical section simultaneously. We then show that the expected value of $\Phi(C_t)$ is proportional to L_t, and in particular that it is small if few `lock()` calls have finished. This gives a bound on the probability that two processes are in the critical section using Markov's inequality.

```
Class Lock(Γ).

shared:
  Boolean Register $S_0, \dots, S_{\Gamma-1}$
  Register $A$ initially $(\perp, 0)$

local:
  Integers $i, seq = 0$ ($seq$ has global scope)

Method lock()
1  $i = 0$
2  while $i < \Gamma$ do
3      Choose random $rnd \in \{R, W\}$ s.t. $Prob(rnd = W) = \frac{1}{2}$
4      if $rnd = R$ then
5          if $S_i = 1$ then
6              $i = 0$;
7          end
8      else
9          $S_i$.write(1); $i = i + 1$
10     end
11     if $i = 0$ then
12         $a = A$
13         if $S_0 = 1$ then
14             while $A = a$ do "nothing" done
15         end
16     end
17 end

Method unlock()
18 $i = \Gamma$
19 while $i > 0$ do
20     $S_{i-1}$.write(0); $i := i - 1$
21 end
22 $A$.write($myID, seq + 1$); $seq = seq + 1$
```

Figure 1 Monte Carlo Mutual Exclusion.

To denote the value of a local variable of a process p, we add subscript p to the variable name. For example, i_p denotes the value of p's local variable i. To bound the probability of error, we define a potential function. The potential of a process $p \in \{1, \dots, n\}$ is

$$\alpha(p) = \begin{cases} -1 & \text{if } p \text{ is poised to read in lines 12-14} \\ & \text{and entered this section through line 6} \\ 2^{i_p} - 1 & \text{otherwise.} \end{cases} \tag{4.1}$$

Hence, $\alpha(p) = -1$ if and only if p is poised in lines 12-14, and prior to entering that section it read $S_{i_p} = 1$ in line 5. The potential of register index $j \in \{0, \dots, \Gamma - 1\}$ is

$$\beta(j) = -S_j \cdot w^j \tag{4.2}$$

Finally, the potential of the system at time t is

$$\Phi(C_t) = \sum_{p=1}^{n} \alpha(p) + \sum_{j=0}^{\Gamma-1} \beta(j) + (n-1) \tag{4.3}$$

▶ **Lemma 4.** *Suppose at some point t_1 process p reads $S_{j_1} = 1$ in line 5, and at a point $t_2 \geq t_1$ it reads $S_{j_2} = 1$ in line 5. Then at some point $t' \in [t_1, t_2]$ either the value of S_0 changes from 0 to 1, or the value of A changes.*

Proof. After reading $S_{j_1} = 1$ at point t_1, process p proceeds to execute line 13, and thus it executes lines 12-14 during $[t_1, t_2]$. Let $t' \in [t_1, t_2]$ be the point when it reads S_0 in line 13.

First assume $S_0 = 1$ at point t'. Then p enters the while-loop in line 14, and does not leave the while-loop until A has changed at least once since p's previous read of A in line 12. Hence, in that case A changes at some point between $[t_1, t_2]$, and the claim is true.

Now assume $S_0 = 0$ at point t'. We show that S_0 changes from 0 to 1 at some point in $[t', t_2]$, which proves the claim. If $j_2 = 0$, then at point t_2 process p reads $S_0 = 1$, so this is obvious. Hence, assume $j_2 > 0$. Then before point t_2 process p must increment its local variable i_p by at least one, which means it writes 1 to S_0 in line 9. ◀

▶ **Lemma 5.** *For a random execution that ends at point t with L `lock()` calls completed,* $\mathrm{E}[\Phi(C_t)] \leq 2n(L+1)$.

Proof. Consider the initial configuration C_0 where each process is poised to begin a `lock()` call and all registers are 0. Then for all processes p, $\alpha(p) = 0$, and for all $j \in \{0, \ldots, \Gamma - 1\}$, $\beta(j) = 0$. Hence, $\Phi(C_0) = n - 1 < n$. We bound the expected value of $\Phi(C_t)$ in subsequent steps by case analysis. Whenever the adversary schedules a process p that has a pending `lock()` or `unlock()` call, p will do one of the following:

(1) Set $S_{i_p} = 0$ in line 20;
(2) Exit Lines 12–14 having entered from line 5;
(3) Exit Lines 12–14 having entered from line 6;
(4) Stay in Lines 12–14;
(5) Choose rnd_p at random in line 3 and then immediately either read S_{i_p} in line 5 or write S_{i_p} in line 9.

We will show that in cases (1), (2), (4), and (5) the expected change in Φ is less than or equal to 0. In case (3) Φ increases by 1. However, case (3) can only occur at most twice per process per successful lock call leading to our bound on $\Phi(C_t)$.

(1) Suppose p sets $S_{i_p-1} = 0$ in line 20. Then $\alpha(p)$ decreases by 2^{i_p-1}. If S_{i_p-1} was 1, then β_{i_p-1} increases by S_{i_p-1} and Φ does not change. If S_{i_p-1} was 0, then Φ decreases.

(2) Next suppose p reads $S_0 = 0$ in line 13 or reads some $A \neq a_p$ in line 14 having entered from line 5 (i.e., $\alpha(p) = 0$). Then p becomes poised to execute line 3 and Φ does not change.

(3) Next suppose p reads $S_0 = 0$ in line 13 or p reads some $A \neq a_p$ in line 14 having entered from line 6$p = R$ (i.e., when $\alpha(p) = -1$). Then no register gets written, p's local variable i remains at value 0, and p ceases to be poised to execute a line in the range 12-14, so $\alpha(p)$ increases from -1 to 0. So Φ increases by 1.

(4) Next, suppose p reads $S_0 = 1$ in line 13, reads A in line 12, or reads $A = a_p$ in line 14. Then no register gets written, $\alpha(p)$ does not change, and p's local variable i remains at 0, so Φ stays the same.

(5) Finally, suppose that when p gets scheduled it chooses rnd_p at random, and then it either reads or writes S_{i_p}, depending on its random choice rnd_p. First assume $S_{i_p} = 0$ when p gets scheduled. If $rnd_p = R$, then p reads S_{i_p} in line 5, and becomes poised to either

read A in line 12 (if $i_p = 0$) entering this section from line 5 so $\alpha(p)$ does not change, or becomes poised to choose rnd_p at random again in line 3. In either case Φ does not change. If $rnd_p = W$, then p proceeds to write 1 to S_{i_p} in line 9, increments its local variable i to $i'_p = i_p + 1$, and either enters the critical section (if $i'_p = \Gamma$), or becomes poised to make another random choice in line 3. Hence, the value of $\alpha(p)$ increases by 2^{i_p} (from $2^{i_p} - 1$ to $2^{i_p+1} - 1$). Since S_{i_p} changes from 0 to 1, the value of $\beta(i_p)$ decreases by 2^{i_p}. Therefore, the change of potential Φ is 0.

Now suppose $S_{i_p} = 1$ when p gets scheduled. If $rnd_p = R$, then p reads $S_{i_p} = 1$ in line 5, and then immediately sets i to 0, and becomes poised to read A in line 12 entering from line 6. Thus, p's new potential is -1. No register gets written, so Φ changes by the same amount as $\alpha(p)$, which is -2^{i_p}. If $rnd_p = W$, then p writes 1 to S_{i_p} in line 5, then increments its local variable i to $i'_p = i_p + 1$, and either enters the critical section if $i'_p = \Gamma$, or become poised to make another random choice in line 3. Hence, p's potential increases by 2^{i_p}. To summarize, if $S_{i_p} = 1$, then with probability 1/2 the value of Φ increases by 2^{i_p}, and with probability 1/2 it decreases by 2^{i_p}. Therefore the expected value of Φ does not change in this case.

The only time Φ can increase in expectation is in case (3), in which case it increases by 1. We will now show that for any process p, this case can happen at most twice per critical section. Case (3) can only occur by entering lines 12–14 by reading $S_{i_p} = 1$ in line 5.

By Lemma 4 we have that if process p reads $S_{i_p} = 1$ at t_1 and $t_2 > t_1$, then the value of S_0 changes from 0 to 1 or the value of A changes at some point $t' \in [t_1, t_2]$. Let U_t be the number of completed `unlock()` calls, L_t be the number of completed `lock()` calls, and A_t be the value of A_{seq} at time t. Since A_{seq} is only incremented at the end of a completed lock call, $A_t \le U_t$. Since an unlock call is preceded by a successful `lock()` call, $U_t \le L_t$. Hence $A_t \le L_t$. The number of times S_0 changes from 0 to 1 is also bounded by one more than the number of completed `lock()` calls at time t. Value 0 is written to S_0 only once per `unlock()` call. Thus the number of times S_0 changes from 0 to 1 is at most $1 + U_t \le 1 + L_t$, and at any time t, the number of times a process p has taken a step of type (3) is at most $1 + 2L_t$. We thus have

$$\mathrm{E}\left[\Phi(C_t)\right] \le \Phi(C_0) + n(1 + 2L) = (n-1) + n + 2nL < 2n(L+1). \tag{4.4}$$

◀

▶ **Lemma 6.** *In any execution, at any point there exists at least one process p_{max} with local variable $i_{p_{max}}$ such that $S_j = 0$ for all $j \in \{i_{p_{max}}, \ldots, \Gamma - 1\}$.*

Proof. Consider any point t during an execution of the mutual exclusion algorithm. Let p_{max} be a process such that $i_{p_{max}}$ is maximal at that point. For the purpose of contradiction assume there is an index $j \in \{i_{p_{max}}, \ldots, \Gamma - 1\}$, such that $S_j = 1$ at point t. Let p' be the last process that writes 1 to S_j at some point $t' \le t$. I.e.,

$$S_j = 1 \text{ throughout } (t', t]. \tag{4.5}$$

Moreover, when p' writes 1 to S_j in line 9 at point t', $i_{p'} = j$, and immediately after writing it increments $i_{p'}$ to $j + 1$. Since $i_{p'} \le i_{p_{max}} \le j$ at point t, process p' must at some later point $t^* \in (t', t)$ decrement $i_{p'}$ from $j+1$ to j. This can only happen when p' executes line 20 while $i_{p'} = j + 1$. But then p' also writes 0 to S_j at $t^* \in (t', t)$, which contradicts (4.5). ◀

▶ **Lemma 7.** *In any reachable configuration C, $\Phi(C)$ is non-negative.*

Proof. By Lemma 6 there exists a process p_{max} such that $S_j = 0$ for all $j \in \{i_{p_{max},\ldots,\Gamma-1}\}$. Then

$$\alpha(p_{max}) = 2^{i_{p_{max}}} - 1 = \sum_{j=0}^{i_{p_{max}-1}} 2^j \geq \sum_{j=0}^{\Gamma-1} S_j \cdot 2^j = -\sum_{j=0}^{\Gamma-1} \beta(j).$$

Since $\alpha(p) \geq -1$ for each other process p,

$$\Phi(C) = n - 1 + \sum_p \alpha(p) + \sum_{j=0}^{\Gamma-1} \beta(j) \geq n - 1 + \sum_{p \neq p_{max}} \alpha(p) \geq n - 1 + \sum_{p \neq p_{max}} -1 = 0.$$ ◀

▶ **Lemma 8.** *If C is a configuration in which at least two processes are in the critical section, $\Phi(C) \geq 2^\Gamma$.*

Proof. Suppose that in C, distinct processes p_1 and p_2 are in the critical section. Then $\alpha(p_1) = \alpha(p_2) = 2^\Gamma - 1$. Since $\alpha(p) \geq -1$ for each other process, and $\beta(j) \geq -2^j$

$$\Phi(C) \geq \big(2(2^\Gamma - 1) + (n-2) \cdot (-1)\big) + \left(\sum_{j=0}^{\Gamma-1} -2^j\right) + (n-1) = 2^\Gamma \tag{4.6}$$

◀

▶ **Lemma 9.** *For $\Gamma = c \log n$, the probability that mutual exclusion is violated at any point before L `lock()` calls finish is $O\left(L^2 \cdot n^{-c+1}\right)$.*

Proof. Let t_j for $j \in \{2, \ldots, L\}$ be the point when the j-th `lock()` call completes. By Lemma 5, $E[\Phi(C_{t_j})] = O(n \cdot j)$, so by Lemmas 7, 8 and Markov's inequality,

$$\Pr\left[C_{t_j} \in \mathcal{C}_{fail}\right] \leq \Pr\left[\Phi(C_{t_j}) \geq 2^\Gamma\right] = O\left(\frac{n \cdot j}{2^\Gamma}\right).$$

Mutual exclusion is violated before L `lock()` calls finish if and only if it is violated after ℓ `lock()` calls finish for some $\ell \in \{2, \ldots, L-1\}$. The probability of that event is

$$\begin{aligned}\Pr\left[\exists j \in \{2, \ldots, L-1\} : C_{t_j} \in \mathcal{C}_{fail}\right] &\leq \Pr\left[\exists j \in \{2, \ldots, L-1\} : \Phi(C_{t_j}) \geq 2^\Gamma\right] \\ &\leq \sum_{j=2}^{L-1} \Pr\left[\Phi(C_{t_j}) \geq 2^\Gamma\right] = O\left(\sum_{j=2}^{L-1} \frac{n(j+1)}{2^\Gamma}\right) = O\left(\frac{n \cdot L^2}{n^c}\right) = O\left(\frac{L^2}{n^{c-1}}\right).\end{aligned}$$

◀

4.2.2 Deadlock-freedom

▶ **Lemma 10.** *The algorithm is randomized deadlock-free.*

Proof. Consider any point t in an infinite fair execution, in which at least one process has a pending `lock()` call. We will show that some process enters the critical section after point t with probability 1.

Suppose no process enters the critical section in $[t, \infty)$. Since `unlock()` is wait-free, there is a point $t_1 \geq t$ such that after t_1 there are no more pending `unlock()` calls. Hence, throughout $[t_1, \infty)$ no process writes 0 to any register S_j, $j \in \{0, \ldots, \Gamma - 1\}$. In other words, only value 1 may get written to any register S_j after point t_1. Since there are only a finite number of registers S_j, there is a point t_2 such that no register S_j, $j \in \{0, \ldots, \Gamma - 1\}$, changes value after t_2. By Lemma 6 there is a process p_{max} such that at point t_2 we have $S_j = 0$ for all $j \in \{i_{p_{max}}, \ldots, \Gamma - 1\}$. Let i^* be the value of $i_{p_{max}}$ at point t_2. Thus,

$$S_{i^*} = \cdots = S_{\Gamma-1} = 0 \text{ throughout } [t_2, \infty). \tag{4.7}$$

If $i^* > 0$, then at t_2 process p_{max} is not poised to execute a shared memory operation in lines 12-14 (because $i_{p_{max}} = i^*$ at that point). Hence, p_{max} is either poised to read in line 5 or to write in line 9. The latter is not possible, as p_{max} would eventually write 1 to S_{i^*}, contradicting (4.7). If p_{max} reads in line 5, then it reads 0 from $S_{i_{p_{max}}}$, where $i_{p_{max}} = i^* > 0$, and so it will begin another iteration of the while-loop with $i_{p_{max}} = i^*$. Repeating the argument, p_{max} will execute an infinite number of iterations of the outer while-loop, each time choosing at random $rnd = R$, and then reading S_{i^*} in line 5. This event has probability 0.

Hence, consider the case $i^* = 0$. First assume that at some point after t_2 some process p is not poised to execute line 14. Then due to (4.7) the if-condition in line 13 remains false for p throughout $[t_3, \infty)$, so p executes an infinite number of iterations of the outer while-loop. With probability 1 process p will eventually in some iteration choose $rnd = W$ in line 3 and then write 1 to some register S_j, $j \in \{0, \ldots, \Gamma - 1\}$. This contradicts (4.7) since we assumed $i^* = 0$.

Thus, throughout $[t_2, \infty)$ all processes with pending `lock()` calls are stuck in the inner while-loop in line 14. Consider any process q stuck in the while-loop, and let T be the point when it read A for last time prior to becoming stuck. Let a^* be the value of A at point T. Register A gets only written in line 22, and due to the increasing sequence number, the same value never gets written twice. Hence, since q is stuck in line 14, it reads $A = a^*$ infinitely many times, and thus

$$\text{no process writes } A \text{ throughout } [T, \infty). \tag{4.8}$$

But at some point $T_1 > T$ and before q becomes stuck in the while-loop, it reads $S_0 = 1$ in line 13. By (4.7), after T_1 some process writes 0 to S_0, and then it will eventually write to A. This contradicts (4.8). ◀

4.2.3 RMR Bound

▶ **Lemma 11.** *In an execution with L invoked* `lock()` *calls, the expected total number of RMRs is $O\big((n + \Gamma)L\big)$.*

The remainder of this section is devoted to the proof of this lemma.

Let $X_{p,\ell}$ denote the number of RMRs a process p incurs in line ℓ, where ℓ is one of 5, 9, 12, 13, 14, 20, and 22. These are the only lines where a process executes shared memory operations, so the total number of RMRs is obtained by summing over all $X_{p,\ell}$.

We now consider a random execution, and condition on the event that the random execution contains L `lock()` calls.

▶ **Lemma 12.** *For each $j \in \{0, \ldots, \Gamma - 1\}$, each process incurs in total at most $L + 1$ RMRs by reading value 0 from register S_j.*

Proof. Value 0 is written to S_j (in line 20) only once per `unlock()` call. Only the first read by a process in the execution, or the first read following such a write of value 0 can at the same time return 0 and incur an RMR. Now the claimed bound follows from the fact that there are at most L `lock()`, and thus at most L `unlock()` calls. ◀

▶ **Lemma 13.** *For any process p we have*

$$X_{p,12} + X_{p,14} \le L + 1 \textit{ and } X_{p,13} \le 2(L + 1).$$

Proof. The value of A changes at most once per `unlock()` call, and thus at most L times during execution E. Hence, process p incurs at most $L + 1$ RMRs by reading A. This proofs the claimed upper bound on $X_{p,12} + X_{p,14}$.

By Lemma 12, the number of RMRs incurred by p's reads of value 0 in line 13 is at most $L + 1$. If process p reads 1 from S_0 in that line, then, due to the while-loop in line 14, it does not read S_0 again until A changed at least once since p's preceding read of A in line 12. In particular, for each read of value 1 from S_0, there is a distinct RMR incurred by p when reading A in line 14. Hence, $X_{p,13} \le L + 1 + X_{p,14} \le 2(L + 1)$. ◀

▶ **Lemma 14.** $E[\sum_p X_{p,9}] = O\big((n + \Gamma)L)\big)$

Proof. For $b \in \{0, 1\}$ let Z_b denote the number of times a write in line 9 (by any process) overwrites value b with value 1. Thus,

$$\sum_p X_{p,9} = Z_0 + Z_1. \tag{4.9}$$

Since each register S_j is reset to 0 only once per `unlock()` call, it can change from 0 to 1 at most $L + 1$ times. Accounting for Γ registers, we obtain

$$Z_0 \le \Gamma(L + 1). \tag{4.10}$$

Now suppose $S_j = 1$ when process p makes a random choice in line 3. With probability $1 - 1/w$ process p decides to read, and if it does so, it reads $S_j = 1$. Hence, p overwrites in 9 a register that has value 1 in expectation at most $1/(1 - 1/w) - 1 = 1/(w - 1)$ times before p reads a register with value 1 in line 5. By Lemma 4 between any two such reads, either S_0 changes from 0 to 1 or A changes, and each of these events happens at most once per `unlock()` call. Thus, the expected number of times process p writes to a register S_j that has value 1 is at most $1 + 1/(w - 1) \cdot L$. Summing over all processes we obtain $E[Z_1] = O(n \cdot L)$ (recall that $Z_0 = Z_1 = 0$ if $L = 0$). Now the claim follows from (4.9) and (4.10). ◀

▶ **Lemma 15.** *For any process p we have*

$$E[X_{p,5}] = O\big((n + \Gamma)L\big).$$

Proof. Let Y_p denote the number of times process p reads a value of 1 in line 5. By Lemma 4, between any two such consecutive reads, either the value of A changes, or S_0 changes from 0 to 1. Since S_0 can change from 1 to 0 at most L times (once for each `unlock()` call), it can change from 0 to 1 at most $L + 1$ times. The value of A can also change at most once for each `unlock()` call, and thus at most L times. Hence, $Y_p \le 2L + 2$.

By Lemma 12 process p incurs at most $L + 1$ RMRs by reading value 0 from S_0 in line 5. Now suppose $j > 0$. Then p reads S_j only after writing 1 to S_{j-1} in line 9, which contributes to $X_{p,5}$. Because p chooses to write S_j (instead of reading it) with probability $1/w$, the expected number of times p can read S_j in consecutive iterations of the while-loop (and thus before changing i_p) is at most $w - 1$. Hence, for all x,

$$E[X_{p,5} \,|\, X_{p,9} = x] \le E[Y_p \,|\, X_{p,9} = x] + L + 1 + x(w - 1) \le 3(L + 1) + x(w - 1)$$

Summing this conditional expectation weighted with $\Pr[X_{p,9} = x]$ over all values of x, yields

$$E[X_{p,5}] \le 3L + 3 + E[X_{p,9}] \cdot (w - 1).$$

Now the claim follows from Lemma 14. ◀

Lemma 11 now follows from Lemmas 12, 13, and 14:

Proof of Lemma 11. If $L = 0$, then no process calls `lock()` or `unlock()`, so the lemma is trivially true. Hence, we assume w.l.o.g. that $L \geq 1$. Since there are at most L `unlock()` calls in total, we have

$$\sum_p X_{p,20} \leq \Gamma \cdot L \text{ and } \sum_p X_{p,22} \leq L. \qquad ◀$$

4.3 When the algorithm fails

Because the algorithm is randomized, there is a nonzero chance that it violates mutual exclusion even in short executions. We can guard against this using Lamport's fast mutual exclusion algorithm [15], which is now often abstracted in the form of a **splitter** object [16]. Lamport's fast mutual exclusion algorithm uses $O(1)$ space and $O(1)$ time to either allow a process into the critical section or deny it entry, and works as long as at most one process at a time invokes it. Because our algorithm guarantees mutual exclusion for polynomially many critical sections with high probability, in most executions we will not see multiple processes attempting to access the Lamport mutex, and so each process will successfully acquire this mutex. In the even that a process does not acquire the Lamport mutex, then our algorithm has failed; the process can then unlock the randomized algorithm and move over to a backup algorithm to attempt to acquire a mutex there. A 2-process mutex algorithm (using $O(1)$ space and $O(1)$ time) can then be used to choose between processes leaving the Lamport mutex and the backup mutex.

Because the combined mutex never uses more than $O(n)$ objects, the high-probability $O(\log n)$ space bound also gives a bound on expected space. The full result is:

▶ **Corollary 16.** *There is a randomized mutual exclusion algorithm with expected amortized RMR complexity $O(n)$, such that the algorithm is randomized deadlock-free; satisfies mutual exclusion in all executions; uses at most $O(n)$ objects in all executions; and, for a polynomial number of* `lock()` *calls, uses $O(\log n)$ objects in expectation and with high probability.*

5 Simulating allocation on update

With a more refined space complexity measure comes the need to develop new tools for minimizing this measure. In this section, we describe a technique for designing protocols where the space complexity is proportional to the number of objects that are **updated** as opposed to all objects that are accessed. We distinguish between **update** operations that can change an object's state and **read** operations that cannot; an object is considered to be updated if an update operation is applied to it, even if its state is not changed by this particular application.

Counting only updates corresponds to an **allocate-on-update** model where merely reading an object costs nothing. We show that this model gives costs equivalent up to a factor logarithmic in size of the address space to the allocate-on-use model of Definition 1.

To obtain this result, we construct a data structure where the objects $O_1, O_2, \ldots$ are the leaves of a binary search tree whose internal nodes are one-bit registers that record if any object in their subtree has been updated. A read operation on some object O_i starts at the root of the tree and follows the path to O_i until it sees a 0, indicating that O_i can be treated as still being in its initial state, or reaches O_i and applies the operation to it. Conversely, an update operation starts by updating O_i and then sets all the bits in order along the path from O_i to the root.

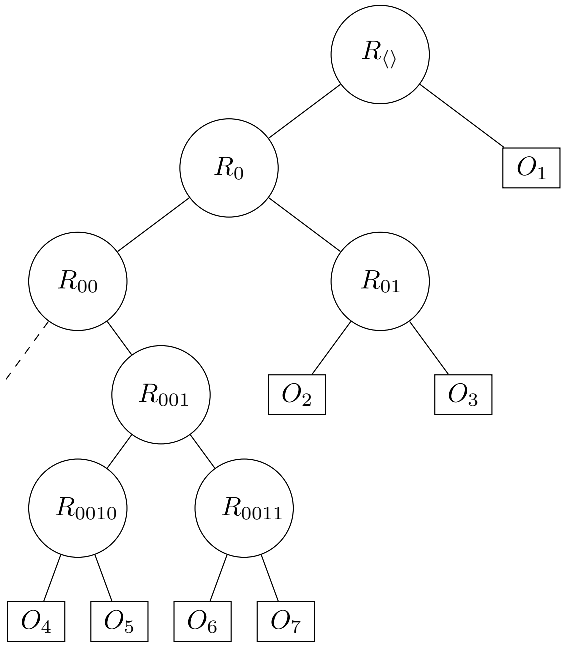

Figure 2 Tree derived from Elias gamma code.

The structure of the tree is based on the well-known correspondence between binary trees and prefix-free codes. Here the left edge leaving each node is labeled with a 0 and the right edge with a 1, the path to each leaf gives a code word, and the path to each internal node gives a proper prefix of a code word. The particular code we will use to construct the tree is the **Elias gamma code** [9]. This encodes each positive integer i as a sequence of bits, by first expressing i as its unique binary expansion $1i_1i_2\dots i_n$, and then constructing a codeword $\gamma(i) = 0^n1i_1i_2\dots i_n$. This gives a codeword for each positive integer i with length $2\lfloor \lg i \rfloor + 1 = O(\log i)$. The first few levels of the resulting tree are depicted in Figure 2.

Pseudocode for the simulation is given in Algorithm 4. Each register is labeled by a codeword prefix. The objects are labeled with their original indices.

▶ **Lemma 17.** *Algorithm 4 gives a linearizable implementation of $O_1, O_2, \dots$, such that in any execution in which update operations start on at most m objects, and the maximum index of these objects is s:*

1. *The space complexity is $O(m \log s)$.*
2. *The step complexity of an* `apply`(π) *operation where π is an update to O_i is $O(\log i)$.*
3. *The step complexity of an* `apply`(π) *operation where π is a read of O_i is $O(\min(\log i, \log s))$.*

Proof. We start by showing linearizability.

Given a concurrent execution H of Algorithm 4, we will construct an explicit linearization S. The first step in this construction is to assign a linearization point to each operation π in H. If π is an update operation on some object O_i, its linearization point is the first step in H at which (a) π has been applied to O_i, and (b) every bit in an ancestor of O_i is set. If π is a read operation, its linearization point is the step at which either π is applied to O_i, or the process executing π reads a 0 from an ancestor of O_i. In the case of an update operation π, the linearization point follows the step in which π is applied to O_i and precedes the return of π (since π cannot return without setting all ancestors of O_i to 1). In the case of a read operation π, the linearization point corresponds to an actual step of π. In both cases, the linearization point of π lies within π's execution interval.

Algorithm 4: Applying operation π to object O_i.

```
procedure apply($\pi$)
    Let $O_i$ be the object on which $\pi$ is an operation
    Let $x_1 x_2 \dots x_k$ be the encoding of $i$
    if $\pi$ is an update then
        $r \leftarrow \pi(O_i)$
        for $j \leftarrow k-1 \dots 0$ do
            $R_{x_1 \dots x_j} \leftarrow 1$
        end
        return $r$
    else
        for $j \leftarrow 0 \dots k-1$ do
            if $R_{x_1 \dots x_j} = 0$ then
                return $\pi$ applied to the initial state of $O_i$
            end
        end
        // Reached only if all nodes on path are 1
        return $\pi(O_i)$
    end
end
```

If we declare $\rho \leq \sigma$ whenever ρ's linearization point precedes σ's, we get a preorder on all operations in H. Because each operation's lineaerization point lies within its execution interval, this preorder is consistent with the observed execution order in H. But it is not necessarily a total order because update operations that are applied to the same object O_i may be assigned the same linearization point: the first step at which all ancestors of O_i are 1. Should this occur, we break ties by ordering such updates according to the order in which they were applied to O_i. We now argue that the resulting total order gives a sequential execution S on $O_1, O_2, \dots$. This requires showing that each operation that returns in H returns the same value in H as it would in S.

Fix some particular O_i. The operations on O_i can be divided into three groups:

1. Read operations that observe 0 in an ancestor of O_i.
2. Update operations that are applied to O_i before all ancestors of O_i are 1.
3. Read or update operations that are applied to O_i after all ancestors of O_i are 1.

That these groups include all operations follows from the fact that any update operation is applied either before or after all ancestors of O_i are 1, and any read operation that does not observe a 0 will eventually be applied to O_i after all ancestors of O_i are 1.

Now observe that all operations in the first group are assigned linearization points before the step at which all ancestors of O_i are 1; in the second group, at this step; and in the third group, after this step. So S restricted to O_i consists of a group of read operations that return values obtained from the inital state of O_i, consistent with having no preceding updates; followed by a sequence of updates linearized in the same order that they are applied to O_i in H; followed by a sequence that may contain mixed updates and reads that are again linearized in the same order that they are applied to O_i in H. Since the first group of operations contain only read operations, the operations applied to O_i in H start with the same initial state as in S, and since they are the same operations applied in the the same order, they return the same values.

For space complexity, observe that any object accessed in the execution is either (a) an object O_i that is updated; (b) a register that is the ancestor of an object that is updated; or (c) a register or object all of whose ancestors are set to 1. Since a register is set to 1 only if it is an ancestor of an updated object, and since each such register has at least one child that is either in category (a) or (b), there is an injection from the set of registers and objects in category (c) to their parents in category (b). Category (a) requires m space; (b) requires $O(m \log s)$ space; and thus (c) also requires $O(m \log s)$ space. This gives $O(m \log s)$ space total.

Time complexity of updates is immediate from the code; `apply`(π) traverses $O(\log i)$ nodes to reach O_i. For reads, `apply`(π) follows a path of length $O(\log i)$ that stops early if it reaches a node not on the path to an updated object; since any such path to an updated object has length $O(\log s)$, this gives the claimed bound. ◀

We believe that these overheads are the best possible using a binary tree structure. However, using a tree with higher arity (equivalent to using a code with a larger alphabet) could produce a lower time complexity overhead at the cost of more wasted space. We do not have a lower bound demonstrating that this particular trade-off is necessary, so the exact complexity of simulating allocate-on-update in the simpler allocate-on-access model remains open.

6 Open problems

While we have started a formal approach to analyzing allocate-on-use space complexity for shared-memory distributed algorithms, much remains to be done.

We have demonstrated that it is possible to solve mutual exclusion for a polynomial number of locks with logarithmic space complexity with high probability. Our algorithm pays for its low space complexity with linear RMR complexity. Curiously, it is possible to achieve both $O(1)$ space and RMR complexity with high probability using very long random delays under the assumption that critical sections are not held for long; this follows from Lamport's fast mutual exclusion algorithm [15] and is essentially a randomized version of the delay-based algorithm of Fischer described by Lamport. However, this algorithm has poor step complexity even in the absence of contention. We conjecture that there exists a randomized algorithm for mutual exclusion that simultaneously achieves $O(\log n)$ space complexity, $O(\log n)$ RMR complexity, and $O(\log n)$ uncontended step complexity, all with high probability assuming polynomially many passages through the critical section.

We have also shown that a system that assumes an allocate-on-update model can be simulated in the stricter allocate-on-access model with a logarithmic increase in the number of objects used. It is not clear whether this overhead is necessary, or whether it could be eliminated with a more sophisticated simulation.

References

1 Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Distributed Computing: 25th International Symposium, DISC 2011*, volume 6950 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, 2011.

2 Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th International Symposium, DISC 2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings*, volume 6343 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2010. doi:10.1007/978-3-642-15763-9_9.

3 James Aspnes. Faster randomized consensus with an oblivious adversary. In *2012 ACM Symposium on Principles of Distributed Computing*, pages 1–8, 2012.

4 James Aspnes and Faith Ellen. Tight bounds for adopt-commit objects. *Theory of Computing Systems*, 55(3):451–474, 2014. doi:10.1007/s00224-013-9448-1.

5 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 217–226, 2008. doi:10.1145/1374376.1374410.

6 Hagit Attiya, Fabian Kuhn, C. Greg Plaxton, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient adaptive collect using randomization. *Distributed Computing*, 18(3):179–188, 2006. doi:10.1007/s00446-005-0143-6.

7 Michael A. Bender and Seth Gilbert. Mutual exclusion with o(log^2 log n) amortized work. In Rafail Ostrovsky, editor, *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 728–737. IEEE Computer Society, 2011. doi:10.1109/FOCS.2011.84.

8 James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993. doi:10.1006/inco.1993.1065.

9 P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975. doi:10.1109/TIT.1975.1055349.

10 Rati Gelashvili. On the optimal space complexity of consensus for anonymous processes. In Yoram Moses, editor, *Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 452–466, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-48653-5_30.

11 George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 19–28. ACM, 2012. doi:10.1145/2332432.2332436.

12 George Giakkoupis and Philipp Woelfel. Randomized abortable mutual exclusion with constant amortized RMR complexity on the CC model. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 221–229. ACM, 2017. doi:10.1145/3087801.3087837.

13 Danny Hendler and Philipp Woelfel. Randomized mutual exclusion with sub-logarithmic rmr-complexity. *Distributed Computing*, 24(1):3–19, 2011. doi:10.1007/s00446-011-0128-6.

14 Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for non-blocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000. doi:10.1137/S0097539797317299.

15 Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987. doi:10.1145/7351.7352.

16 Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.

17 Jae-Heon Yang and James H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995. doi:10.1007/BF01784242.

18 Leqi Zhu. A tight space bound for consensus. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 345–350. ACM, 2016. doi:10.1145/2897518.2897565.