A Time Complexity Bound for Adaptive Mutual Exclusion*

(Extended Abstract)

Yong-Jik Kim and James H. Anderson

Department of Computer Science
University of North Carolina at Chapel Hill

Abstract. We consider the time complexity of adaptive mutual exclusion algorithms, where "time" is measured by counting the number of remote memory references required per critical-section access. We establish a lower bound that precludes a deterministic algorithm with $O(\log k)$ time complexity (in fact, any deterministic $o(k)$ algorithm), where k is "point contention." In contrast, we show that expected $O(\log k)$ time *is* possible using randomization.

1 Introduction

In this paper, we consider the time complexity of adaptive mutual exclusion algorithms. A mutual exclusion algorithm is *adaptive* if its time complexity is a function of the number of contending processes [3, 6, 8, 10, 11]. Under the time complexity measure considered in this paper, only remote memory references that cause a traversal of the global processor-to-memory interconnect are counted. Specifically, we count the number of such references generated by one process p in a computation that starts when p becomes active (leaves its noncritical section) and ends when p once again becomes inactive (returns to its noncritical section). Unless stated otherwise, we let k denote the "point contention" over such a computation (the *point contention* over a computation H is the maximum number of processes that are active at the same state in H [1]). Throughout this paper, we let N denote the number of processes in the system.

In recent work, we presented an adaptive mutual exclusion algorithm — henceforth called Algorithm AK — with $O(min(k, \log N))$ time complexity [3]. Algorithm AK requires only read/write atomicity and is the only such algorithm known to us that is adaptive under the remote-memory-references time complexity measure. In other recent work, we established a worst-case time bound of $\Omega(\log N / \log \log N)$ for mutual exclusion algorithms (adaptive or not) based on reads, writes, or comparison primitives such as test-and-set and compare-and-swap [4]. (A *comparison primitive* conditionally updates a shared variable after first testing that its value meets some condition.) This result shows

* Work supported by NSF grants CCR 9732916, CCR 9972211, CCR 9988327, and ITR 0082866.

that the $O(\log N)$ worst-case time complexity of ALGORITHM AK is close to optimal. In fact, we believe it *is* optimal: we conjecture that $\Omega(\log N)$ is a tight lower bound for this class of algorithms.

If $\Omega(\log N)$ is a tight lower bound, then presumably a lower bound of $\Omega(\log k)$ would follow as well. This suggests two interesting possibilities: in all likelihood, either $\Omega(min(k, \log N))$ is a *tight* lower bound (*i.e.*, ALGORITHM AK is optimal), or it is possible to design an adaptive algorithm with $O(\log k)$ time complexity (*i.e.*, $\Omega(\log k)$ is tight). Indeed, the problem of designing an $O(\log k)$ algorithm using only reads and writes has been mentioned in two recent papers [3, 6].

In this paper, we show that an $O(\log k)$ algorithm in fact does not exist. In particular, we prove the following: *For any k, there exists some N such that, for any N-process mutual exclusion algorithm based on reads, writes, or comparison primitives, a computation exists involving $\Theta(k)$ processes in which some process performs $\Omega(k)$ remote memory references to enter and exit its critical section.*

Although this result precludes a deterministic $O(\log k)$ algorithm (in fact, any deterministic $o(k)$ algorithm), we show that a randomized algorithm does exist with *expected* $O(\log k)$ time complexity. This algorithm is obtained through a simple modification to ALGORITHM AK.

The rest of the paper is organized as follows. In Sec. 2, our system model is defined. Our lower bound proof is presented in Secs. 3-4. The radomized algorithm mentioned above is sketched in Sec. 5. We conclude in Sec. 6.

2 Definitions

Our model of a shared-memory system is based on that used in [4, 5].

Shared-memory systems. A *shared-memory system* $\mathcal{S} = (C, P, V)$ consists of a set of computations C, a set of processes P, and a set of variables V. A *computation* is a finite sequence of events.

An *event* e is denoted $[R, W, p]$, where $p \in P$. The sets R and W consist of pairs (v, α), where $v \in V$. This notation represents an event of process p that reads the value α from variable v for each element $(v, \alpha) \in R$, and writes the value α to variable v for each element $(v, \alpha) \in W$. Each variable in R (or W) is assumed to be distinct. We define $Rvar(e)$, the set of variables read by e, to be $\{v \mid (v, \alpha) \in R\}$, and $Wvar(e)$, the set of variables written by e, to be $\{v \mid (v, \alpha) \in W\}$. We also define $var(e)$, the set of all variables accessed by e, to be $Rvar(e) \cup Wvar(e)$. We say that this event *accesses* each variable in $var(e)$, and that process p is the *owner* of e, denoted $owner(e) = p$. For brevity, we sometimes use e_p to denote an event owned by process p.

Each variable is *local* to at most one process and is *remote* to all other processes. (Note that we allow variables that are remote to *all* processes.) An *initial value* is associated with each variable. An event is *local* if it does not access any remote variable, and is *remote* otherwise.

We use $\langle e, \ldots \rangle$ to denote a computation that begins with the event e, and $\langle\rangle$ to denote the empty computation. We use $H \circ G$ to denote the computation

obtained by concatenating computations H and G. The value of variable v at the end of computation H, denoted $value(v, H)$, is the last value written to v in H (or the initial value of v if v is not written in H). The last event to write to v in H is denoted $writer_event(v, H)$, and its owner is denoted $writer(v, H)$. (Although our definition of an event allows multiple instances of the same event, we assume that such instances are distinguishable from each other.) If v is not written by any event in H, then we let $writer(v, H) = \bot$ and $writer_event(v, H) = \bot$.

For a computation H and a set of processes Y, $H \mid Y$ denotes the subcomputation of H that contains all events in H of processes in Y. Computations H and G are *equivalent* with respect to Y iff $H \mid Y = G \mid Y$. A computation H is a *Y-computation* iff $H = H \mid Y$. For simplicity, we abbreviate the preceding definitions when applied to a singleton set of processes. For example, if $Y = \{p\}$, then we use $H \mid p$ to mean $H \mid \{p\}$ and p-computation to mean $\{p\}$-computation.

The following properties apply to any shared-memory system.

- **(P1)** If $H \in C$ and G is a prefix of H, then $G \in C$.
- **(P2)** If $H \circ \langle e_p \rangle \in C$, $G \in C$, $G \mid p = H \mid p$, and if $value(v, G) = value(v, H)$ holds for all $v \in Rvar(e_p)$, then $G \circ \langle e_p \rangle \in C$.
- **(P3)** If $H \circ \langle e_p \rangle \in C$, $G \in C$, $G \mid p = H \mid p$, then $G \circ \langle e'_p \rangle \in C$ for some event e'_p such that $Rvar(e'_p) = Rvar(e_p)$ and $Wvar(e'_p) = Wvar(e_p)$.
- **(P4)** For any $H \in C$, $H \circ \langle e_p \rangle \in C$ implies that $\alpha = value(v, H)$ holds, for all $(v, \alpha) \in R$, where $e_p = [R, W, p]$.

For notational simplicity, we make the following assumption, which requires each remote event to be either an atomic read or an atomic write.

Atomicity Assumption: Each event of a process p may either read or write (but not both) at most one variable that is remote to p. □

As explained later, this assumption actually can be relaxed to allow comparison primitives.

Mutual exclusion systems. We now define a special kind of shared-memory system, namely mutual exclusion systems, which are our main interest.

A *mutual exclusion system* $\mathcal{S} = (C, P, V)$ is a shared-memory system that satisfies the following properties. Each process $p \in P$ has a local variable $stat_p$ ranging over $\{ncs,\ entry,\ exit\}$ and initially ncs. $stat_p$ is accessed only by the events $Enter_p = [\{\},\ \{(stat_p, entry)\},\ p]$, $CS_p = [\{\},\ \{(stat_p, exit)\},\ p]$, and $Exit_p = [\{\},\ \{(stat_p, ncs)\},\ p]$, and is updated only as follows: for all $H \in C$,

$H \circ \langle Enter_p \rangle \in C$ iff $value(stat_p, H) = ncs$;
$H \circ \langle CS_p \rangle \in C$ only if $value(stat_p, H) = entry$;
$H \circ \langle Exit_p \rangle \in C$ only if $value(stat_p, H) = exit$.

(Note that $stat_p$ transits directly from *entry* to *exit*.)

In our proof, we only consider computations in which each process enters and then exits its critical section at most once. Thus, we henceforth assume that each computation contains at most one $Enter_p$ event for each process p. The remaining requirements of a mutual exclusion system are as follows.

Exclusion: For all $H \in C$, if both $H \circ \langle CS_p \rangle \in C$ and $H \circ \langle CS_q \rangle \in C$ hold, then $p = q$.

Progress (starvation freedom): For all $H \in C$, if $value(stat_p, H) \neq ncs$, then there exists an X-computation G such that $H \circ G \circ \langle e_p \rangle \in C$, where $X = \{q \in P \mid value(stat_q, H) \neq ncs\}$ and e_p is either CS_p (if $value(stat_p, H) = entry$) or $Exit_p$ (if $value(stat_p, H) = exit$). □

Cache-coherent systems. On cache-coherent shared-memory systems, some remote variable accesses may be handled without causing interconnect traffic. Our lower-bound proof applies to such systems without modification. This is because we do not count every remote event, but only critical events, as defined below.

Definition 1. *Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system. Let e_p be an event in $H \in C$. Then, we can write H as $F \circ \langle e_p \rangle \circ G$, where F and G are subcomputations of H. We say that e_p is a* critical event *in H iff one of the following conditions holds:*

State transition event: *e_p is one of $Enter_p$, CS_p, or $Exit_p$.*

Critical read: *There exists a variable v, remote to p, such that $v \in Rvar(e_p)$ and $F \mid p$ does not contain a read from v.*

Critical write: *There exists a variable v, remote to p, such that $v \in Wvar(e_p)$ and $writer(v, F) \neq p$.* □

Note that state transition events do *not* actually cause cache misses; these events are defined as critical events because this allows us to combine certain cases in the proofs that follow. A process executes only three transition events per critical-section execution, so this does not affect our asymptotic lower bound.

According to Definition 1, a remote read of v by p is critical if it is the first read of v by p. A remote write of v by p is critical if **(i)** it is the first write of v by p (which implies that either $writer(v, F) = q \neq p$ holds or $writer(v, F) = \perp \neq p$ holds); or **(ii)** some other process has written v since p's last write of v (which also implies that $writer(v, F) \neq p$ holds).

Note that if p both reads and writes v, then both its first read of v and first write of v are considered critical. Depending on the system implementation, the latter of these two events might not generate a cache miss. However, even in such a case, the first such event will always generate a cache miss, and hence at least half of all such critical reads and writes will actually incur real global traffic. Hence, our lower bound remains asymptotically unchanged for such systems.

In a *write-through* cache scheme, writes always generate a cache miss. With a *write-back* scheme, a remote write to a variable v may create a cached copy of v, so that subsequent writes to v do not cause cache misses. In Definition 1, if e_p is not the first write to v by p, then it is considered critical only if $writer(v, F) = q \neq p$ holds, which implies that v is stored in the local cache line of another process q. (Effectively, we are assuming an idealized cache of infinite size: a cached variable may be updated or invalidated but it is never replaced by another variable. Note that $writer(v, F) = q$ implies that q's cached copy of v has not

been invalidated.) In such a case, e_p must either invalidate or update the cached copy of v (depending on the system), thereby generating global traffic.

Note that the definition of a critical event depends on the particular computation that contains the event, specifically the prefix of the computation preceding the event. Therefore, when saying that an event is (or is not) critical, the computation containing the event must be specified.

3 Proof Strategy

In Sec. 4, we show that for any positive k, there exists some N such that, for any mutual exclusion system $\mathcal{S} = (C, P, V)$ with $|P| \geq N$, there exists a computation H such that some process p experiences point contention k and executes at least k critical events to enter and exit its critical section. The proof focuses on a special class of computations called "regular" computations. A regular computation consists of events of two groups of processes, "active processes" and "finished processes." Informally, an active process is a process in its entry section, competing with other active processes; a finished process is a process that has executed its critical section once, and is in its noncritical section. (These properties follow from (R4), given later in this section.)

Definition 2. *Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system, and H be a computation in C. We define* $\mathrm{Act}(H)$*, the set of* active processes *in H, and* $\mathrm{Fin}(H)$*, the set of* finished processes *in H, as follows.*

$$\begin{aligned} \mathrm{Act}(H) &= \{p \in P \mid H \,|\, p \neq \langle\rangle \text{ and } \langle \mathit{Exit}_p \rangle \text{ is not in } H\} \\ \mathrm{Fin}(H) &= \{p \in P \mid H \,|\, p \neq \langle\rangle \text{ and } \langle \mathit{Exit}_p \rangle \text{ is in } H\} \end{aligned}$$ □

The proof proceeds by inductively constructing longer and longer regular computations, until the desired lower bound is attained. The regularity condition defined below ensures that *no participating process has knowledge of any other process that is active*. This has two consequences: **(i)** we can "erase" any active process (*i.e.*, remove its events from the computation) and still get a valid computation; **(ii)** "most" active processes have a "next" critical event. In the definition that follows, (R1) ensures that active processes have no knowledge of each other; (R2) and (R3) bound the number of possible conflicts caused by appending a critical event; (R4) ensures that the active and finished processes behave as explained above; (R5) ensures that the property of being a critical write is conserved when considering certain related computations.

Definition 3. *Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system, and H be a computation in C. We say that H is* regular *iff the following conditions hold.*

- **(R1)** *For any event e_p and f_q in H, where $p \neq q$, if p writes to a variable v, and if another process q reads* that *value from v, then $p \in \mathrm{Fin}(H)$.*
- **(R2)** *If a process p accesses a variable that is local to another process q, then $q \notin \mathrm{Act}(H)$.*

(R3) *For any variable v, if v is accessed by more than one processes in* $\mathrm{Act}(H)$, *then either* $writer(v, H) = \perp$ *or* $writer(v, H) \in \mathrm{Fin}(H)$ *holds.*

(R4) *For any process p that participates in H ($H \,|\, p \neq \langle\rangle$),* $value(stat_p, H)$ *is entry, if* $p \in \mathrm{Act}(H)$*, and ncs otherwise (i.e.,* $p \in \mathrm{Fin}(H)$*). Moreover, if* $p \in \mathrm{Fin}(H)$*, then the last event of p in H is* $Exit_p$.

(R5) *Consider two events e_p and f_p such that e_p precedes f_p in H, both e_p and f_p write to a variable v, and f_p is a critical write to v in H. In this case, there exists a write to v by some process r in* $\mathrm{Fin}(H)$ *between e_p and f_p.* □

Proof overview. Initially, we start with a regular computation H_1, where $\mathrm{Act}(H_1) = P$, $\mathrm{Fin}(H_1) = \{\}$, and each process has exactly one critical event. We then inductively show that other longer computations exist, the last of which establishes our lower bound. Each computation is obtained by rolling some process forward to its noncritical section (NCS) or by erasing some processes — this basic proof strategy has been used previously to prove several other lower bounds for concurrent systems [2, 4, 7, 12]. We assume that P is large enough to ensure that enough non-erased processes remain after each induction step for the next step to be applied. The precise bound on $|P|$ is given in Theorem 2.

At the jth induction step, we consider a computation H_j such that $\mathrm{Act}(H_j)$ consists of n processes that execute j critical events each. We construct a regular computation H_{j+1} such that $\mathrm{Act}(H_{j+1})$ consists of $\Omega(\sqrt{n}/k)$ processes that execute $j + 1$ critical events each. The construction method, formally described in Lemma 4, is explained below. In constructing H_{j+1} from H_j, some processes may be erased and *at most one* rolled forward. At the end of step $k-1$, we have a regular computation H_k in which each active process executes k critical events and $|\,\mathrm{Fin}(H_k)| \leq k-1$. Since active processes have no knowledge of each other, a computation involving at most k processes can be obtained from H_k by erasing all but one active process; the remaining process performs k critical events.

We now describe how H_{j+1} is constructed from H_j. We show in Lemma 3 that, among the n processes in $\mathrm{Act}(H_j)$, at least $n-1$ can execute an additional critical event prior to its critical section. We call these events "future" critical events, and denote the corresponding set of processes by Y. We consider two cases, based on the variables remotely accessed by these future critical events.

Erasing strategy. Assume that $\Omega(\sqrt{n})$ distinct variables are remotely accessed by the future critical events. For each such variable v, we select one process whose future critical event accesses v, and erase the rest. Let Y' be the set of selected processes. We now eliminate any information flow among processes in Y' by constructing a "conflict graph" $\mathcal{G}$ as follows.

Each process p in Y' is considered a vertex in $\mathcal{G}$. By induction, process p has j critical events in $\mathrm{Act}(H_j)$ and one future critical event. An edge (p, q), where $p \neq q$, is included in $\mathcal{G}$ **(i)** if the future critical event of p remotely accesses a local variable of process q, or **(ii)** if one of p's $j + 1$ critical events accesses the same variable as the future critical event of process q.

Since each process in Y' accesses a distinct remote variable in its future critical event, it is clear that each process generates at most one edge by rule (i)

and at most $j+1$ edges by rule (ii). By applying Turán's theorem (Theorem 1), we can find a subset Z of Y' such that $|Z| = \Omega(\sqrt{n}/j)$ and their critical events do not conflict with each other. By retaining Z and erasing all other active processes, we can eliminate all conflicts. Thus, we can construct H_{j+1}.

Roll-forward strategy. Assume that the number of distinct variables that are remotely accessed by the future critical events is $O(\sqrt{n})$. Since there are $\Theta(n)$ future critical events, there exists a variable v that is remotely accessed by future critical events of $\Omega(\sqrt{n})$ processes. Let Y_v be the set of these processes. First, we retain Y_v and erase all other active processes. Let the resulting computation be H'. We then arrange the future critical events of Y_v by placing all writes before all reads. In this way, the only information flow among processes in Y_v is that from the "last writer" of v to all the subsequent readers (of v). Let p_{LW} be the last writer. We then roll p_{LW} forward by generating a regular computation G from H' such that $\mathrm{Fin}(G) = \mathrm{Fin}(H') \cup \{p_{\mathrm{LW}}\}$.

If p_{LW} executes at least k critical events before reaching its NCS, then the $\Omega(k)$ lower bound easily follows. Therefore, we can assume that p_{LW} performs fewer than k critical events while being rolled forward. Each critical event of p_{LW} that is appended to H' may generate information flow only if it reads a variable v that is written by another process in H'. Condition (R3) guarantees that if there are multiple processes that write to v, the last writer in H' is not active. Because information flow from an inactive process is allowed, a conflict arises only if there is a single process that writes to v in H'. Thus, each critical event of p_{LW} conflicts with at most one process in Y_v, and hence can erase at most one process. (Appending a noncritical event to H' cannot cause any processes to be erased. In particular, if a noncritical remote read by p_{LW} is appended, then p_{LW} must have previously read the same variable. By (R3), if the last writer is another process, then that process is not active.)

Therefore, the entire roll-forward procedure erases fewer than k processes from $\mathrm{Act}(H') = Y_v$. We can assume $|P|$ is sufficiently large to ensure that $\sqrt{n} > 2k$. This ensures that $\Omega(\sqrt{n})$ processes survive after the entire procedure. Thus, we can construct H_{j+1}.

4 Lower Bound for Systems with Read/Write Atomicity

In this section, we present our lower-bound theorem for systems satisfying the Atomicity Assumption. At the end of this section, we explain why the lower bound also holds for systems with comparison primitives. We begin by stating several lemmas. Lemma 1 states that we can safely "erase" any active process. Lemma 2 allows us to extend a computation by noncritical events. Lemma 3 is used to show that "most" active processes have a "next" critical event.

Lemma 1. *Consider a regular computation H in C. For any set Y of processes such that* $\mathrm{Fin}(H) \subseteq Y$*, the following hold:* $H \,|\, Y \in C$*,* $H \,|\, Y$ *is regular,* $\mathrm{Fin}(H \,|\, Y) = \mathrm{Fin}(H)$*, and an event e in $H \,|\, Y$ is a critical event iff it is also a critical event in H.* □

Lemma 2. *Consider a regular computation H in C, and a set of processes $Y = \{p_1,\ p_2,\ \ldots,\ p_m\}$, where $Y \subseteq \mathrm{Act}(H)$. Assume that for each p_j in Y, there exists a p_j-computation L_{p_j}, such that $H \circ L_{p_j} \in C$ and L_{p_j} has no critical events in $H \circ L_{p_j}$. Define L to be $L_{p_1} \circ L_{p_2} \circ \cdots \circ L_{p_m}$. Then, the following hold: $H \circ L \in C$, $H \circ L$ is regular,* $\mathrm{Fin}(H \circ L) = \mathrm{Fin}(H)$, *and L has no critical events in $H \circ L$.* □

Lemma 3. *Let H be a regular computation in C. Define $n = |\,\mathrm{Act}(H)|$. Then, there exists a subset Y of* $\mathrm{Act}(H)$, *where $n-1 \leq |Y| \leq n$, satisfying the following: for each process p in Y, there exist a p-computation L_p and an event e_p of p such that*

- *$H \circ L_p \circ \langle e_p \rangle \in C$;*
- *L_p contains no critical events in $H \circ L_p$;*
- *$e_p \notin \{Enter_p, CS_p, Exit_p\}$;*
- *e_p is a critical event of p in $H \circ L_p \circ \langle e_p \rangle$;*
- *$H \circ L_p$ is regular;*
- $\mathrm{Fin}(H \circ L_p) = \mathrm{Fin}(H)$. □

The next theorem by Turán [13] will be used in proving Lemma 4.

Theorem 1 (Turán). *Let $\mathcal{G} = (V, E)$ be an undirected graph, where V is a set of vertices and E is a set of edges. If the average degree of $\mathcal{G}$ is d, then there exists an independent set*[1] *with at least $\lceil |V|/(d+1) \rceil$ vertices.* □

The following lemma provides the induction step of our lower-bound proof.

Lemma 4. *Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system, k be a positive integer, and H be a regular computation in C. Define $n = |\,\mathrm{Act}(H)|$. Assume that $n > 1$ and*

- *each process in* $\mathrm{Act}(H)$ *executes exactly c critical events in H.* (1)

Then, one of the following propositions is true.

(Pr1) *There exist a process $p \in \mathrm{Act}(H)$ and a computation $F \in C$ such that*

- *$F \circ \langle Exit_p \rangle \in C$;*
- *F does not contain $\langle Exit_p \rangle$;*
- *at most m processes participate in F, where $m = |\,\mathrm{Fin}(H) + 1|$;*
- *p executes at least k critical events in F.*

(Pr2) *There exists a regular computation G in C such that*

- $\mathrm{Act}(G) \subseteq \mathrm{Act}(H)$*;*
- $|\,\mathrm{Fin}(G)| \leq |\,\mathrm{Fin}(H) + 1|$*;*
- $|\,\mathrm{Act}(G)| \geq \min(\sqrt{n}/(2c+3),\ \sqrt{n} - k)$*;* (2)
- *each process in* $\mathrm{Act}(G)$ *executes exactly $c+1$ critical events in G.*

Proof. Because H is regular, using Lemma 3, we can construct a subset Y of $\mathrm{Act}(H)$ such that

$$n - 1 \leq |Y| \leq n, \tag{3}$$

and for each $p \in Y$, there exist a p-computation L_p and an event e_p such that

[1] An *independent set* of a graph $\mathcal{G} = (V, E)$ is a subset $V' \subseteq V$ such that no edge in E is incident to two vertices in V'.

- $H \circ L_p \circ \langle e_p \rangle \in C$; (4)
- L_p contains no critical events in $H \circ L_p$; (5)
- $e_p \notin \{Enter_p, CS_p, Exit_p\}$; (6)
- e_p is a critical event of p in $H \circ L_p \circ \langle e_p \rangle$; (7)
- $H \circ L_p$ is regular; (8)
- $\mathrm{Fin}(H \circ L_p) = \mathrm{Fin}(H)$. (9)

Define V_{fut} as the set of variables remotely accessed by the "future" critical events:

$$V_{\mathrm{fut}} = \{v \in V \mid \text{there exists } p \in Y \text{ such that } e_p \text{ remotely accesses } v\}. \quad (10)$$

We consider two cases, depending on the size of V_{fut}.

Case 1: $|V_{\mathrm{fut}}| \geq \sqrt{n}$ **(erasing strategy)** By definition, for each variable v in V_{fut}, there exists a process p in Y such that e_p remotely accesses v. Therefore, we can arbitrarily select one such process for each variable v in V_{fut} and construct a subset Y' of Y such that

- if $p \in Y'$, $q \in Y'$ and $p \neq q$, then e_p and e_q access different remote variables, and (11)
- $|Y'| = |V_{\mathrm{fut}}| \geq \sqrt{n}$. (12)

We now construct a graph $\mathcal{G} = (Y', E_{\mathcal{G}})$, where each vertex is a process in Y'. To each process y in Y', we apply the following rules.

(G1) Let $v \in V_{\mathrm{fut}}$ be the variable remotely accessed by e_y. If v is local to a process z in Y', then introduce edge (y, z).

(G2) For each critical event f of y in H, let v_f be the variable remotely accessed by f. If $v_f \in V_{\mathrm{fut}}$ and v_f is remotely accessed by event e_z for some process $z \neq y$ in Y', then introduce edge (y, z).

Because each variable is local to at most one process, and since an event can access at most one remote variable, (G1) can introduce at most one edge per process. Since, by (1), y executes exactly c critical events in H, by (11), (G2) can introduce at most c edges per process.

Combining (G1) and (G2), at most $c+1$ edges are introduced per process. Thus, the average degree of $\mathcal{G}$ is at most $2(c+1)$. Hence, by Theorem 1, there exists an independent set $Z \subseteq Y'$ such that

- $|Z| \geq |Y'|/(2c+3) \geq \sqrt{n}/(2c+3)$, (13)

where the latter inequality follows from (12).

Next, we construct a computation G, satisfying (Pr2), such that $\mathrm{Act}(G) = |Z|$. Let $H' = H \mid (Z \cup \mathrm{Fin}(H))$. For each process $z \in Z$, (4) implies $H \circ L_z \in C$. Hence, by (8) and (9), and applying Lemma 1 with 'H' $\leftarrow H \circ L_z$ and 'Y' $\leftarrow Z \cup \mathrm{Fin}(H)$, we have the following:

- $H' \circ L_z \in C$ (which, by (P1), implies $H' \in C$), and
- an event in $H' \circ L_z$ is critical iff it is also critical in $H \circ L_z$. (14)

By (5), the latter also implies that L_z contains no critical events in $H' \circ L_z$.

Let $m = |Z|$ and index the processes in Z as $Z = \{z_1,\ z_2,\ \ldots,\ z_m\}$. Define $L = L_{z_1} \circ L_{z_2} \circ \cdots \circ L_{z_m}$. By applying Lemma 2 with 'H' $\leftarrow H'$ and 'Y' $\leftarrow Z$, we have the following:

- $H' \circ L \in C$,
- $H' \circ L$ is regular, and
- L contains no critical events in $H' \circ L$. (15)

By the definition of H' and L, we also have

- for each $z \in Z$, $(H' \circ L) \mid z = (H \circ L_z) \mid z$. (16)

Therefore, by (4) and Property (P3), for each $z_j \in Z$, there exists an event e'_{z_j}, such that

- $G \in C$, where $G = H' \circ L \circ E$ and $E = \langle e'_{z_1},\ e'_{z_2},\ \ldots,\ e'_{z_m} \rangle$;
- $\mathit{Rvar}(e'_{z_j}) = \mathit{Rvar}(e_{z_j})$, $\mathit{Wvar}(e'_{z_j}) = \mathit{Wvar}(e_{z_j})$, and $\mathit{owner}(e'_{z_j}) = \mathit{owner}(e_{z_j}) = z_j$.

Conditions (R1)–(R5) can be individually checked to hold in G, which implies that G is a regular computation. Since $Z \subseteq Y' \subseteq Y \subseteq \mathrm{Act}(H)$, by (1), (14), and (15), each process in Z executes exactly c critical events in $H' \circ L$.

We now show that every event in E is critical in G. Note that, by (7), e_z is a critical event in $H \circ L_z \circ \langle e_z \rangle$. By (6), e_z is not a transition event. By (16), the events of z are the same in both $H \circ L_z$ and $H' \circ L$. Thus, if e_z is a critical read or a "first" critical write in $H \circ L_z \circ \langle e_z \rangle$, then it is also critical in G. The only remaining case is that e_z writes some variable v remotely, and is critical in $H \circ L_z \circ \langle e_z \rangle$ because of a write to v prior to e_z by another process not in G. However, (R5) ensures that in such a case there exists some process in $\mathrm{Fin}(H)$ that writes to v before e_z, and hence e_z is also critical in G.

Thus, we have constructed a computation G that satisfies the following: $\mathrm{Act}(G) = Z \subseteq \mathrm{Act}(H)$, $\mathrm{Fin}(G) = \mathrm{Fin}(H') = \mathrm{Fin}(H)$ (from the definition of H', and since $L \circ E$ does not contain transition events), $|\,\mathrm{Act}(G)| \geq \sqrt{n}/(2c+3)$ (from (13)), and each process in $\mathrm{Act}(G)$ executes exactly $c+1$ critical events in G (from the preceding paragraph). It follows that G satisfies (Pr2).

Case 2: $|V_{\mathrm{fut}}| \leq \sqrt{n}$ (roll-forward strategy) For each variable v_j in V_{fut}, define $Y_{v_j} = \{p \in Y \mid e_p \text{ remotely accesses } v_j\}$. By (3) and (10), $|V_{\mathrm{fut}}| \leq \sqrt{n}$ implies that there exists a variable v_j in V_{fut} such that $|Y_{v_j}| \geq (n-1)/\sqrt{n}$ holds. Let v be one such variable. Then, the following holds:

$$|Y_v| \geq (n-1)/\sqrt{n} > \sqrt{n} - 1. \tag{17}$$

Define $H' = H \mid (Y_v \cup \mathrm{Fin}(H))$. Using $Y_v \subseteq Y \subseteq \mathrm{Act}(H)$, we also have

$$\mathrm{Act}(H') = Y_v \subseteq \mathrm{Act}(H) \ \wedge\ \mathrm{Fin}(H') = \mathrm{Fin}(H). \tag{18}$$

Because H is regular, by Lemma 1,

- $H' \in C$, (19)
- H' is regular, and (20)
- an event in H' is a critical event iff it is also a critical event in H. (21)

We index processes in Y_v from y_1 to y_m, where $m = |Y_v|$, such that if e_{y_i} writes to v and e_{y_j} reads v, then $i < j$ (*i.e.*, future writes to v precede future reads from v).

For each $y \in Y_v$, let $F_y = (H \circ L_y) \mid (Y_v \cup \mathrm{Fin}(H))$. (4) implies $H \circ L_y \in C$. Hence, by (8), and applying Lemma 1 with 'H' $\leftarrow H \circ L_y$ and 'Y' $\leftarrow Y_v \cup \mathrm{Fin}(H)$,

we have the following: $F_y \in C$, and an event in F_y is critical iff it is also critical in $H \circ L_y$. Since $y \in Y_v$ and L_y is a y-computation, by the definition of H', $F_y = H' \circ L_y$. Hence, by (5), we have

- $H' \circ L_y \in C$, and (22)
- L_y does not have a critical event in $H' \circ L_y$. (23)

Define $L = L_{y_1} \circ L_{y_2} \circ \cdots \circ L_{y_m}$. We now use Lemma 2, with 'H' $\leftarrow H'$ and 'Y' $\leftarrow Y_v$. The antecedent of the lemma follows from (18), (19), (20), (22), and (23). This gives us the following.

- $H' \circ L \in C$,
- $H' \circ L$ is regular, (24)
- $\mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H)$, and (25)
- L contains no critical events in $H' \circ L$. (26)

By the definition of H' and L, we also have

- for each $y \in Y_v$, $(H' \circ L) \,|\, y = (H \circ L_y) \,|\, y$. (27)

Therefore, by (4) and Property (P3), for each $y_j \in Y_v$, there exists an event e'_{y_j}, such that

- $\overline{G} \in C$, where $\overline{G} = H' \circ L \circ E$ and $E = \langle e'_{y_1},\ e'_{y_2},\ \ldots,\ e'_{y_m} \rangle$;
- $\mathit{Rvar}(e'_{y_j}) = \mathit{Rvar}(e_{y_j})$, $\mathit{Wvar}(e'_{y_j}) = \mathit{Wvar}(e_{y_j})$, and $\mathit{owner}(e'_{y_j}) = \mathit{owner}(e_{y_j}) = y_j$.

From (6) and (26), it follows that $L \circ E$ does not contain any transition events. Moreover, by the definition of L and E, $(L \circ E) \,|\, p \neq \langle\rangle$ implies $p \in Y_v$, for each process p. Combining these assertions with (18), we have

$$\mathrm{Act}(\overline{G}) = \mathrm{Act}(H' \circ L) = \mathrm{Act}(H') = Y_v \ \wedge$$
$$\mathrm{Fin}(\overline{G}) = \mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H') = \mathrm{Fin}(H). \tag{28}$$

We now claim that each process in Y_v $(= \mathrm{Act}(\overline{G}))$ executes exactly $c+1$ critical events in $\overline{G}$. In particular, by (1), (18), (21), and (26), it follows that each process in Y_v executes exactly c critical events in $H' \circ L$. On the other hand, by (7), e_y is a critical event in $H \circ L_y \circ \langle e_y \rangle$. By (27), and using an argument that is similar to that at the end of Case 1, we can prove that each event e'_y in E is a critical event in $\overline{G}$.

Let p_{LW} be the last process to write to v in $\overline{G}$ (if such a process exists). If p_{LW} does not exist or if $p_{\mathrm{LW}} \in \mathrm{Fin}(\overline{G}) = \mathrm{Fin}(H)$, conditions (R1)–(R5) can be individually checked to hold in $\overline{G}$, which implies that $\overline{G}$ is a regular computation. In this case, (17) and (28) imply that $\overline{G}$ satisfies (Pr2).

Therefore, assume $p_{\mathrm{LW}} \in \mathrm{Act}(\overline{G}) = Y_v$. Define $H_{\mathrm{LW}} = (H' \circ L) \,|\, (\mathrm{Fin}(H) \cup \{p_{\mathrm{LW}}\})$. By (24), (25), and applying Lemma 1 with 'H' $\leftarrow H' \circ L$ and 'Y' $\leftarrow \mathrm{Fin}(H') \cup \{p_{\mathrm{LW}}\}$, we have: $H_{\mathrm{LW}} \in C$, H_{LW} is regular, and $\mathrm{Act}(H_{\mathrm{LW}}) = \{p_{\mathrm{LW}}\}$.

Since p_{LW} is the only active process in H_{LW}, by the Progress property, there exists a p_{LW}-computation F such that $H_{\mathrm{LW}} \circ F \circ \langle \mathit{Exit}_{p_{\mathrm{LW}}} \rangle \in C$ and F contains exactly one $\mathit{CS}_{p_{\mathrm{LW}}}$ and no $\mathit{Exit}_{p_{\mathrm{LW}}}$. If F contains k or more critical events in $H_{\mathrm{LW}} \circ F$, then $H_{\mathrm{LW}} \circ F$ satisfies (Pr1). Therefore, we can assume that F contains at most $k-1$ critical events in $H_{\mathrm{LW}} \circ L$. Let V_{LW} be the set of variables remotely accessed by these critical events.

If a process q in Y_v writes to a variable in V_{LW} in H', it might generate information flow between q and p_{LW}. Therefore, define K, the set of processes to erase (or "kill"), as $K = \{p \in Y_v - \{p_{\mathrm{LW}}\} \mid p = \mathit{writer}(u, H')$ or u is local to p for some $u \in V_{\mathrm{LW}}\}$. (R2) ensures that each variable in V_{LW} introduces at most one process into K. Thus, we have $|K| \le |V_{\mathrm{LW}}| \le k-1$.

Define S, the "survivors," as $S = Y_v - K$, and let $H_S = (H' \circ L) \mid (\mathrm{Fin}(H) \cup S)$. By (24), and applying Lemma 1 with 'H' $\leftarrow H' \circ L$ and 'Y' $\leftarrow \mathrm{Fin}(H') \cup S$, we have the following: $H_S \in C$, H_S is regular, and $\mathrm{Act}(H_S) = S$. By the definition of L, we also have $H_S \mid y = (H' \circ L) \mid y$, for each $y \in S$. Since every event in E accesses only local variables and v, by (P2), we have $H_S \circ E \in C$.

Note that (P2) also implies that the first event of F is $e'_{p_{\mathrm{LW}}}$. Hence, we can write F as $e'_{p_{\mathrm{LW}}} \circ F'$. Define $G = H_S \circ E \circ F'$. Note that $(H_S \circ E) \mid p_{\mathrm{LW}} = (H_{\mathrm{LW}} \mid p_{\mathrm{LW}}) \circ \langle e'_{p_{\mathrm{LW}}} \rangle$, and that events of F' cannot read any variable written by processes in S other than p_{LW} itself. Therefore, by inductively applying (P2) over the events of F', we have $G \in C$.

Conditions (R1)–(R5) can be individually checked to hold in G, which implies that G is a regular computation such that $\mathrm{Fin}(G) = \mathrm{Fin}(H) \cup \{p_{\mathrm{LW}}\}$. Moreover, by (17) and from $|K| \le k-1$, we have $|\mathrm{Act}(G)| = |S| \ge (\sqrt{n}-1)-(k-1) \ge \sqrt{n}-k$. Thus, G satisfies (Pr2). □

Theorem 2. *Let $N(k) = (2k+1)^{2(2^k-1)}$. For any mutual exclusion system $\mathcal{S} = (C, P, V)$ and for any positive number k, if $|P| \ge N(k)$, then there exists a computation H such that at most k processes participate in H and some process p executes at least k critical operations in H to enter and exit its critical section.*

Proof. Let $H_1 = \langle \mathit{Enter}_1,\ \mathit{Enter}_2,\ \ldots,\ \mathit{Enter}_N \rangle$, where $P = \{1, 2, \ldots, N\}$ and $N \ge N(k)$. By the definition of a mutual exclusion system, $H_1 \in C$. It is obvious that H_1 is regular and each process in $\mathrm{Act}(H) = P$ has exactly one critical event in H_1. Starting with H_1, we repeatedly apply Lemma 4 and construct a sequence of computations $H_1,\ H_2,\ \ldots$ such that each process in $\mathrm{Act}(H_j)$ has j critical events in H_j. We repeat the process until either H_k is constructed or some H_j satisfies (Pr1) of Lemma 4.

If some H_j $(j < k)$ satisfies (Pr1), then consider the first such j. By our choice of j, each of $H_1,\ \ldots,\ H_{j-1}$ satisfies (Pr2) of Lemma 4. Therefore, since $|\mathrm{Fin}(H_1)| = 0$, we have $|\mathrm{Fin}(H_j)| \le j-1 < k$. It follows that computation $F \circ \langle \mathit{Exit}_p \rangle$, generated by applying Lemma 4 to H_j, satisfies Theorem 2.

The remaining possibility is that each of $H_1,\ \ldots,\ H_{k-1}$ satisfies (Pr2). We claim that, for $1 \le j \le k$, the following holds:

$$|\mathrm{Act}(H_j)| \ge (2k+1)^{2(2^{k+1-j}-1)}. \tag{29}$$

The induction basis $(j = 1)$ directly follows from $\mathrm{Act}(H) = P$ and $|P| \ge N(k)$. In the induction step, assume that (29) holds for some j $(1 \le j < k)$, and let $n_j = |\mathrm{Act}(H_j)|$. Note that each active process in H_j executes exactly j critical events. By (29), we also have $n_j > 4k^2$, which implies that $\sqrt{n_j} - k > \sqrt{n_j}/2 > \sqrt{n_j}/(2k+1)$. Therefore, by (2), we have $|\mathrm{Act}(H_{j+1})| \ge \min(\sqrt{n_j}/(2j+3),\ \sqrt{n_j} - k) \ge \sqrt{n_j}/(2k+1)$, from which the induction easily follows.

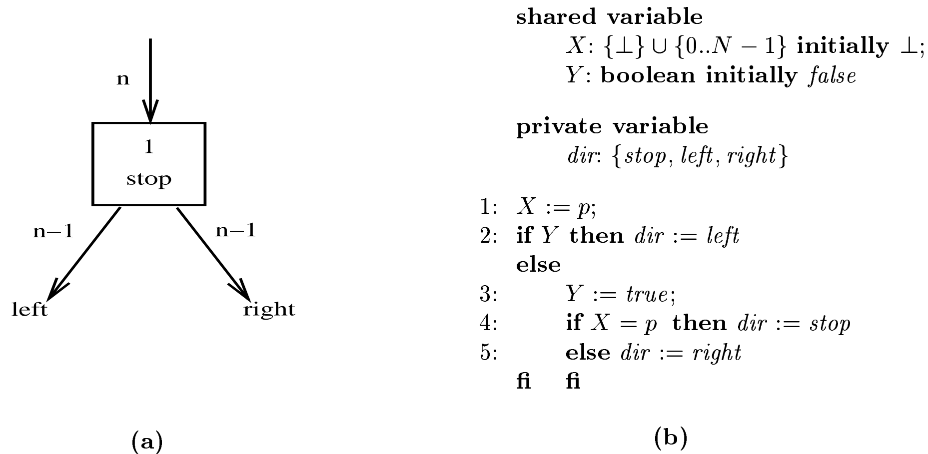

Fig. 1. (a) The splitter element and **(b)** the code fragment that implements it.

Finally, (29) implies $|\,\mathrm{Act}(H_k)| \geq 1$, and (Pr2) implies $|\,\mathrm{Fin}(H_k)| \leq k - 1$. Hence, select any arbitrary process p from $\mathrm{Act}(H_k)$. Define $G = H_k \,|\, (\mathrm{Fin}(H_k) \cup \{p\})$. Clearly, at most k processes participate in G. By applying Lemma 1 with 'H' $\leftarrow H_k$ and 'Y' $\leftarrow \mathrm{Fin}(H_k) \cup \{p\}$, we have the following: $G \in C$, and an event in G is critical iff it is also critical in H_k. Hence, because p executes k critical events in H_k, G is a computation that satisfies Theorem 2. □

Theorem 2 can be easily strengthened to apply to systems in which comparison primitives are allowed. The key idea is this: if several comparison primitives on some shared variable are currently enabled, then they can be applied in an order in which at most one succeeds. A comparison primitive can be treated much like an ordinary write if successful, and like an ordinary read if unsuccessful.

5 Randomized Algorithm

In this section, we describe the randomized version of Algorithm AK mentioned earlier. Due to space constraints, only a high-level description of Algorithm AK is included here. A full description can be found in [3].

At the heart of Algorithm AK is the splitter element from Lamport's fast mutual exclusion algorithm [9]. The splitter element is defined in Fig. 1. Each process that invokes a splitter either stops or moves left or right (as indicated by the value assigned to the variable *dir*). Splitters are useful because of the following properties: if n processes invoke a splitter, then at most one of them can stop at that splitter, and at most $n - 1$ can move left (respectively, right).

In Algorithm AK, splitter elements are used to construct a "renaming tree." A splitter is located at each node of the tree and corresponds to a "name." A process acquires a name by moving down through the tree, starting at the root, until it stops at some splitter. Within the renaming tree is an arbitration

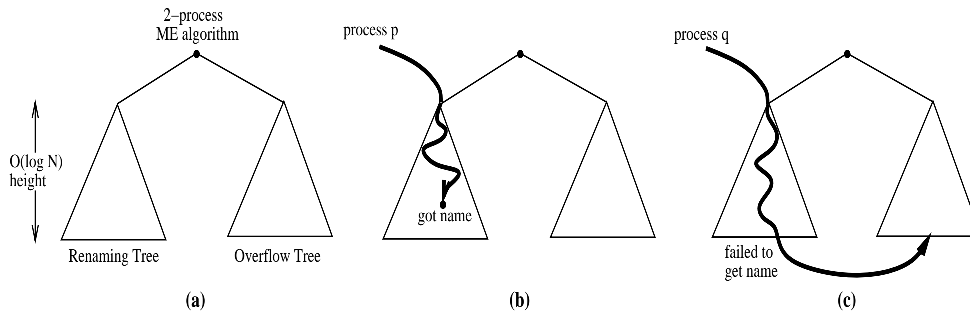

Fig. 2. **(a)** Renaming tree and overflow tree. **(b)** Process p gets a name in the renaming tree. **(c)** Process q fails to get a name and must compete within the overflow tree.

tree that forms dynamically as processes acquire names. A process competes within the arbitration tree by moving up to the root, starting at the node where it acquired its name. Associated with each node in the tree is a three-process mutual exclusion algorithm. As a process moves up the tree, it executes the entry section associated with each node it visits. After completing its critical section, a process retraces its path, this time executing exit sections. A three-process mutual exclusion algorithm is needed at each node to accommodate one process from each of the left- and right-subtrees beneath that node and any process that may have successfully acquired a name at that node.

In ALGORITHM AK, the renaming tree's height is defined to be $\lfloor \log N \rfloor$, which results in a tree with $\Theta(N)$ nodes. With a tree of this height, a process could "fall off" the end of the tree without acquiring a name. To handle such processes, a second arbitration tree, called the "overflow tree," is used. The renaming and overflow trees are connected by placing a two-process mutual exclusion algorithm on top of each tree, as illustrated in Fig. 2.

The time complexity of ALGORITHM AK is determined by the depth to which a process descends in the renaming tree. If the point contention experienced by a process p is k, then the depth to which p descends is $O(k)$. This is because, of the processes that access the same splitter, all but one may move in the same direction from that splitter. If all the required mutual exclusion algorithms are implemented using Yang and Anderson's local-spin algorithm [14], then because the renaming and overflow trees are both of height $\Theta(\log N)$, overall time complexity is $O(min(k, \log N))$.

Our new randomized algorithm is obtained from ALGORITHM AK by replacing the original splitter with a probabilistic version, which is obtained by using "$dir := choice(left,\ right)$" in place of the assignments to *dir* at lines 2 and 5 in Fig. 1(b), where $choice(left, right)$ returns *left* (*right*) with probability 1/2. With this change, a process descends to an expected depth of $\Theta(\log k)$ in the renaming tree. Thus, the algorithm has $\Theta(\log k)$ expected time complexity.

6 Concluding Remarks

We have established a lower bound that precludes an $O(\log k)$ adaptive mutual exclusion algorithm (in fact, any $o(k)$ algorithm) based on reads, writes, or comparison primitives. We have also shown that expected $O(\log k)$ time is possible using randomization.

One may wonder whether a $\Omega(min(k, \log N/\log\log N))$ lower bound follows the results of this paper and [4]. Unfortunately, the answer is no. We have shown that $\Omega(k)$ time complexity is required *provided* N is sufficiently large. This leaves open the possibility that an algorithm might have $\Theta(k)$ time complexity for very "low" levels of contention, but $o(k)$ time complexity for "intermediate" levels of contention. However, we find this highly unlikely.

References

1. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–103. May 1999.
2. Y. Afek, P. Boxer, and D. Touitou. Bounds on the shared memory requirements for long-lived and adaptive objects. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 81–89. July 2000.
3. J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43, October 2000.
4. J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. To be presented at the 20th Annual ACM Symposium on Principles of Distributed Computing, August 2001.
5. J. Anderson and J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.
6. H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–100. July 2000.
7. J. Burns and N. Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pages 833–842, 1980.
8. M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.
9. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.
10. M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.
11. E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 159–168. August 1992.
12. E. Styer and G. Peterson. Tight bounds for shared memory symmetric mutual exclusion. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 177–191. August 1989.
13. P. Turán. On an extremal problem in graph theory (in Hungarian). *Mat. Fiz. Lapok*, 48:436–452, 1941.
14. J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.