Nonatomic Mutual Exclusion with Local Spinning*

(Extended Abstract)

James H. Anderson and Yong-Jik Kim
Department of Computer Science
University of North Carolina at Chapel Hill
{anderson, kimy}@cs.unc.edu

ABSTRACT

We present an N-process local-spin mutual exclusion algorithm, based on nonatomic reads and writes, in which each process performs $\Theta(\log N)$ remote memory references to enter and exit its critical section. This algorithm is derived from Yang and Anderson's atomic tree-based local-spin algorithm in a way that preserves its time complexity. No atomic read/write algorithm with better asymptotic worst-case time complexity (under the remote-memory-references measure) is currently known. This suggests that atomic memory is *not* fundamentally required if one is interested in *worst-case* time complexity.

The same cannot be said if one is interested in *fast-path algorithms* (in which contention-free time complexity is required to be $O(1)$) or *adaptive algorithms* (in which time complexity is required to be proportional to the number of contending processes). We show that such algorithms fundamentally require memory accesses to be atomic. In particular, we show that for any N-process nonatomic algorithm, there exists a single-process execution in which the lone competing process executes $\Omega(\log N / \log\log N)$ remote operations to enter its critical section. Moreover, these operations must access $\Omega(\sqrt{\log N / \log\log N})$ distinct variables, which implies that fast and adaptive algorithms are impossible even if caching techniques are used to avoid accessing the processors-to-memory interconnection network.

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

1. INTRODUCTION

This paper is concerned with shared-memory mutual exclusion algorithms based on read and write operations.[1] In work on such algorithms, *nonatomic* and *local-spin* algorithms have received considerable attention. In nonatomic algorithms, variable accesses are assumed to take place over intervals of time, and hence may overlap one another. In contrast, each variable access in an atomic algorithm is viewed as taking place instantaneously. Requiring atomic memory access is tantamount to assuming mutual exclusion in hardware [18]. Thus, mutual exclusion algorithms requiring this are in some sense circular.

In local-spin algorithms, all busy-waiting loops are read-only loops in which only locally-accessible variables are read; a variable is locally accessible if it is stored in a local cache line (possible on a multiprocessor with coherent caches) or stored in a local memory partition (possible on a distributed shared-memory machine). By structuring busy-waiting loops in this way, contention for the processors-to-memory interconnection network can be greatly reduced. Performance studies presented in several papers [9, 13, 23, 30] have shown that local-spin algorithms typically scale well as contention increases, while non-local-spin algorithms do not.

In this paper, we present possibility and impossibility results pertaining to algorithms that are *both* nonatomic *and* use local spinning. Before describing our main contributions, we first give a brief overview of relevant related research on nonatomic and local-spin algorithms.

Nonatomic algorithms. Lamport was the first to point out the circularity inherent in assuming atomic statement execution [18]. He also presented the first nonatomic algorithm, his famous bakery algorithm [18]. Lamport's work on the bakery algorithm was a catalyst for much subsequent work by him on proof formalisms for nonatomic algorithms (*e.g.*, [22]).

The bakery algorithm is not a local-spin algorithm. In addition, it requires unbounded memory. In later work, Lamport presented four other nonatomic algorithms, each with bounded memory [19]. These algorithms differ in the progress and fault-tolerance properties they satisfy. None are local-spin algorithms.

Local-spin algorithms. The earliest local-spin algorithm based only on reads and writes is also the only *nonatomic* local-spin algorithm known to us [4]. This algorithm, due to Anderson, is composed of a collection of constant-time two-process algorithms, which are used to allow each process to compete individually against every other process. The resulting algorithm has $\Theta(N)$ time complexity,[2] where N is the number of processes. The correctness of the nonatomic version of the algorithm is mainly a consequence of the fact that only single-writer, single-reader, single-bit variables are used. With any nonatomic algorithm, overlapping operations that access a common variable are the main concern. In Anderson's algorithm, if two overlapping operations access the same (single-bit) variable, then one is a read and the other is a write. The assumption usually made (and made herein) regarding such overlapping operations is that the read may return any value [18, 20]. Note that if such a write changes the written variable's value, then an overlapping read can be linearized to occur either before or after the write [18]. For example, if a write changes a variable's value from 1 to 0, then an overlapping read that returns 1 (0) can be linearized to occur immediately before (after) the write.[3] In Anderson's algorithm, most writes write new values, and the structure of the algorithm ensures that those writes that do not have no adverse impact.

In later work, Yang and Anderson showed that sub-linear time complexity was possible, at the price of atomic memory. They established this by presenting a $\Theta(\log N)$ algorithm in which instances of an $O(1)$ two-process algorithm are embedded in a binary arbitration tree [30]. Their two-process algorithm, unlike Anderson's, does not require *statically* allocated single-writer, single-reader variables. Using such variables in an arbitration tree is problematic because the participating processes at each node may vary with time. On the other hand, Yang and Anderson's algorithm uses multi-bit variables, which renders it incorrect if variable accesses are nonatomic.

In recent years, there has been much interest in algorithms that are *fast* in the absence of contention [7, 21, 30] or that are *adaptive*, *i.e.*, with time complexity that is proportional to the number of contending processes [5, 10, 12, 27]. In a recent paper [7], we presented a "fast-path" mechanism that improves the contention-free time complexity of Yang and Anderson's algorithm to $O(1)$, without affecting its worst-case time complexity. In another recent paper [5], we presented an extension of this mechanism that results in an adaptive algorithm with $O(\min(k, \log N))$ time complexity, where k is "point contention" [1].

Several prior research efforts on lower bounds are of relevance to this paper. Of particular relevance are lower bounds established by Anderson and Yang, which involve trade-offs between time complexity and write-contention [8]. The *write-contention* of an algorithm is the number of processes that may be simultaneously enabled to write the same variable. Anderson and Yang showed that any algorithm with write-contention w must have a single-process execution in which that process executes $\Omega(\log_w N)$ remote operations for entry into its critical section. Further, among these operations, $\Omega(\sqrt{\log_w N})$ distinct remote variables are accessed. Thus, a trade-off between write-contention and time complexity exists even in systems with coherent caches.

Because a single-process execution is used to establish these bounds, it follows that $\Omega(N)$-writer variables are needed for fast or adaptive algorithms. In other work [3], Alur and Taubenfeld showed that fast (and hence adaptive) algorithms also require variables with $\Omega(\log N)$ bits (*i.e.*, variables large enough to hold at least some fraction of a process identifier).

In other related work [6], we established a lower bound of $\Omega(\log N / \log\log N)$ remote operations for any mutual exclusion algorithm based on reads and writes. This bound has no bearing on fast or adaptive algorithms because it results from an execution that may involve many processes. (In a later related paper [17], adaptive *atomic* algorithms were considered.)

[1] All claims made hereafter are assumed to pertain to this class of algorithms unless otherwise indicated.

[2] We consider the *time complexity* of an algorithm to be the number of remote memory references (*i.e.*, references that require an access of the interconnection network) required by one process to enter and exit its critical section.

[3] Such reasoning must be used with caution; for example, it may be impossible to linearize a *sequence* of such overlapping reads by the same process without reordering them.

Contributions. Given the research reviewed above, two questions immediately come to mind:

- Is it possible to devise a nonatomic local-spin algorithm with $\Theta(\log N)$ time complexity, *i.e.*, that matches the best atomic algorithm known?
- Is it possible to devise a nonatomic algorithm that is fast or adaptive?

Both questions are answered in this paper. We answer the first question in the affirmative by presenting a $\Theta(\log N)$ nonatomic algorithm, which is derived from Yang and Anderson's arbitration-tree algorithm by means of simple transformations. On the other hand, the answer to the second question is negative. We show this by proving that any nonatomic algorithm must have a single-process execution in which that process executes $\Omega(\log N/\log\log N)$ remote operations to enter its critical section. Moreover, these operations must access $\Omega(\sqrt{\log N/\log\log N})$ distinct variables, which implies that fast and adaptive algorithms are impossible even if caching techniques are used to avoid accessing the interconnection network. Some of the techniques used to establish these bounds are taken from earlier papers [6, 8], while others are new, being applicable when memory accesses are nonatomic. Given the prior results summarized above, it follows that any fast or adaptive algorithm necessarily must use $\Omega(\log N)$-bit, $\Omega(N)$-writer variables that are accessed *atomically*.

Organization. The rest of this paper is as follows. In Sec. 2, our nonatomic algorithm is presented. Definitions needed to establish the bounds mentioned above are then given in Sec. 3. Proof sketches for both bounds are given in Sec. 4; detailed proofs can be found in an appendix. We conclude in Sec. 5.

2. NONATOMIC ALGORITHM

As mentioned earlier, our nonatomic algorithm is derived from Yang and Anderson's algorithm. We hereafter denote these two algorithms as Algorithms NA and YA, respectively; both are depicted in Fig. 1. In this figure, "**await** B" is used as a shorhand for "**while** $\neg B$ **do od**," where B is a boolean expression.

Algorithm YA. We begin with a brief, informal description of Algorithm YA. Associated with each node n at height h in the arbitration tree is a two-process mutual exclusion algorithm, whish uses the following variables: $C[n][0]$, $C[n][1]$, $T[n]$, and a subset of $P[h][0], \ldots, P[h][N-1]$. Variable $C[n][0]$ ranges over $\{0, \ldots, N-1, \perp\}$ and is used by a process from the left subtree rooted at n to inform a process from the right subtree of its intent to enter its critical section. Variable $C[n][1]$ is similarly used by processes from the right subtree. Variable $T[n]$ ranges over $\{0, \ldots, N-1\}$ and is used as a tie-breaker in the event that two processes attempt to "acquire" node n at the same time. Ties are broken in favor of the first process to update $T[n]$. Variable $P[h][p]$ is the spin variable used by process p at node n (if it is among the processes that, by the structure of the tree, can access node n).

Loosely speaking, the two-process algorithm at node n works as follows. A process l from the left subtree rooted at n "announces" its arrival at node n by establishing $C[n][0] = l$. It then assigns its identifier l to the tie-breaker variable $T[n]$, and initializes its spin variable $P[h][l]$. If no process from the right-side has attempted to acquire node n, *i.e.*, if $C[n][1] = \perp$ holds when l executes line 8, then process l proceeds directly to the next level of the arbitration tree (or to its critical section if n is the root). Otherwise, if $C[n][1] = r$, where r is some right-side process, then l reads the tie-breaker variable $T[n]$. If $T[n] \neq l$, then process r has updated $T[n]$ *after* process l, so l can enter its critical section (recall that ties are broken in favor of the first process to update $T[n]$). If $T[n] = l$ holds, then either process r executed line 6 before process l, or process r has executed line 5 but not line 6. In the first case, l should wait until r "releases" node n in its exit section, whereas, in the second case, l should be able to proceed past node n. This ambiguity is resolved by having process l execute lines 10–14. Lines 10–11 are executed by process l to release process r in the event that it is waiting for l to update the tie-breaker variable (*i.e.*, r is busy-waiting at node n at line 12). Lines 12–14 are executed by l to determine which process updated the tie-breaker variable first. Note that $P[h][l] \geq 1$ implies that r has already updated the tie-breaker, and $P[h][l] = 2$ implies that l has released node n. To handle these two cases, process l first waits until $P[h][l] \geq 1$ holds (*i.e.*, until r has updated the tie-breaker), re-examines $T[n]$ to see which process updated it last, and finally, if necessary, waits until $P[h][l] = 2$ holds (*i.e.*, until process r releases node n).

After executing its critical section, process l releases node n by establishing $C[n][0] = 0$. If $T[n] = r$, in which case process r is competing at node n, then process l updates $P[h][r]$ so that process r does not block at node n.

To see that Algorithm YA is a local-spin algorithm, note that each process only waits on spin variables dedicated to it. On a distributed shared-memory machine, a process's spin variables can be stored in a local memory partition. On a cache-coherent machine, the first read of a spin variable by a process p in a busy-waiting loop creates a cached copy. All subsequent reads by p until the variable is written by another process are handled in-cache. The algorithm ensures that after such a write, p's busy-waiting loop terminates.

Algorithm NA. In the rest of this section, we consider the problem of converting Algorithm YA into a nonatomic algorithm. The notion of a nonatomic variable that we assume is that captured by Lamport's definition of a *safe register* [20]: a nonatomic read of a variable returns its current value if it does not overlap any write of that variable, and any arbitrary value from the value domain of the variable if it does overlap such a write. These assumptions are sufficient for our purposes, because our final algorithm precludes overlapping writes of the same variable.

The most obvious way to convert Algorithm YA into a nonatomic algorithm is to implement each atomic variable using nonatomic ones by applying wait-free register constructions presented previously [14, 15, 20, 24, 25, 26]. This is in fact the approach we take for the C and T variables. However, if such constructions are applied to implement the P variables, then a read of such a variable necessarily requires that one or more of the underlying nonatomic variables be written (this was proved by Lamport [20]). As a result, the spins in lines 12 and 14 would no longer be local.

As for the C and T variables, the tree structure ensures that $C[n][s]$ can be viewed as a single-writer, single-reader variable, and $T[n]$ as a two-writer, two-reader variable. Hence, $C[n][s]$ can be implemented quite efficiently

ALGORITHM YA (The original algorithm in [30])

```
process p ::    /* 0 ≤ p < N */

const   /* for simplicity, we assume N = 2^L */
    L = log N;    /* (tree depth) + 1 = O(log N) */
    Tsize = 2^L − 1 = N − 1    /* tree size = O(N) */

shared variables
    T: array[1..Tsize] of 0..N − 1;
    C : array[1..Tsize][0, 1] of (0..N − 1, ⊥) initially ⊥;
    P: array[1..L][0..N − 1] of 0..2 initially 0

private variables
    h: 1..L;
    node: 1..Tsize;
    side: 0..1;        /* 0 = left, 1 = right */
    rival: 0..N − 1

while true do
1:  Noncritical Section;
2:  for h := 1 to L do
3:      node := ⌊(N + p)/2^h⌋;
4:      side := ⌊(N + p)/2^(h−1)⌋ mod 2;
5:      C[node][side] := p;
6:      T[node] := p;
7:      P[h][p] := 0;
8:      rival := C[node][1 − side];
9:      if (rival ≠ ⊥ ∧ T[node] = p) then
10:         if P[h][rival] = 0 then
11:             P[h][rival] := 1 fi;
12:         await P[h][p] ≥ 1;
13:         if T[node] = p then
14:             await P[h][p] = 2 fi
        fi
    od;
15: Critical Section;
16: for h := L downto 1 do
17:     node := ⌊(N + p)/2^h⌋;
18:     side := ⌊(N + p)/2^(h−1)⌋ mod 2;
19:     C[node][side] := ⊥;
20:     rival := T[node];
21:     if rival ≠ p then
22:         P[h][rival] := 2 fi
    od
od
```

(a)

ALGORITHM NA (New nonatomic algorithm)

```
process p ::    /* 0 ≤ p < N */

/* all variable declarations are as */
/* in Algorithm YA, except that */
/* P is replaced by the following */

shared variables
    Q1, Q2, R1, R2: array[1..L][0..N − 1] of boolean

private variables
    qtoggle, rtoggle, temp: 0..1

while true do
1:  Noncritical Section;
2:  for h := 1 to L do
3:      node := ⌊(N + p)/2^h⌋;
4:      side := ⌊(N + p)/2^(h−1)⌋ mod 2;
5:      C[node][side] := p;
6:      T[node] := p;
7:      qtoggle := ¬Q1[h][p];
8:      Q2[h][p] := qtoggle;
9:      rtoggle := ¬R1[h][p];
10:     R2[h][p] := rtoggle;
11:     rival := C[node][1 − side];
12:     if (rival ≠ ⊥ ∧ T[node] = p) then
13:         temp := Q2[h][rival];
14:         Q1[h][rival] := temp;
15:         await (Q1[h][p] = qtoggle) ∨
16:               (R1[h][p] = rtoggle);
17:         if T[node] = p then
18:             await R1[h][p] = rtoggle fi
        fi
    od;
19: Critical Section;
20: for h := L downto 1 do
21:     node := ⌊(N + p)/2^h⌋;
22:     side := ⌊(N + p)/2^(h−1)⌋ mod 2;
23:     C[node][side] := ⊥;
24:     rival := T[node];
25:     if rival ≠ p then
26:         temp := R2[h][rival];
27:         R1[h][rival] := temp fi
    od
od
```

(b)

Figure 1: (a) Algorithm YA and (b) its nonatomic variant. In (b), reads and writes of the C and T variables are assumed to be implemented using register constructions.

using the single-writer, single-reader register construction of Haldar and Subramanian [14]. In this construction, eight nonatomic variables are used, each atomic read requires at most four accesses of nonatomic variables, and each atomic write at most seven. $T[n]$ is more problematic, as it is a multi-reader, multi-writer atomic variable. Nonetheless, register constructions are known that can be used to implement such variables from nonatomic variables with time and space complexity that is polynomial in the number of readers and writers [15, 20, 24, 25, 26]. For variable $T[n]$, the number of readers and writers is constant. Thus, it can be implemented using nonatomic variables with constant space and time complexity.

The need for register constructions to implement the T variables can be obviated by slightly modifying the algorithm, using a technique first proposed by Kessels [16]. (For ease of exposition, this is not done in Algorithm NA in Fig. 1(b).) The idea is to replace each $T[n]$ variable by two

single-bit variables $T1[n]$ and $T2[n]$; $T1[n]$ ($T2[n]$) is written by left-side (right-side) processes and read by right-side (left-side) processes at node n. Left-side processes seek to establish $T1[n] = T2[n]$ and right-side processes seek to establish $T1[n] \neq T2[n]$. Ties are broken accordingly. Because $T1[n]$ and $T2[n]$ are both single-writer, single-reader, single-bit variables, it is relatively straightforward to show that this mechanism still works if variable accesses are nonatomic. In fact, this very mechanism is used in the nonatomic algorithm of Anderson [4].

The P variables can be dealt with similarly. In Algorithm YA, the condition $P[h][p] \geq 1$ indicates that process p may proceed past its first **await**, and the condition $P[h][p] = 2$ indicates that p may proceed past its second **await**. Because multi-bit variables are problematic if memory accesses are nonatomic, we implement these conditions using separate variables. In Algorithm NA (see Fig. 1(b)), variables $Q1[h][p]$ and $Q2[h][p]$ are used to implement the first condition, and variables $R1[h][p]$ and $R2[h][p]$ are used to implement the second. The technique used in updating both pairs of variables is similar to that used in Kessels' tie-breaking scheme described above. In particular, process p attempts to establish $Q1[h][p] \neq Q2[h][p] \wedge R1[h][p] \neq R2[h][p]$ in lines 7–10 and waits while this condition continues to hold at lines 15–16 (note that $qtoggle = Q2[h][p] \wedge rtoggle = R2[h][p]$ holds while p continues to wait). A rival process at node n seeks to establish $Q1[h][p] = Q2[h][p]$ at lines 13–14; the effect is similar to lines 10–11 in Algorithm YA. Lines 18 and 26–27 work in a similar way. As with Kessels' tie-breaking scheme, because the new variables being used here are all single-writer, single-reader, single-bit variables, it is relatively straightforward to show that the algorithm is correct even if variable accesses are nonatomic. A complete proof of this will be given in the full paper. This gives us the following theorem.

THEOREM 1. *The mutual exclusion problem can be solved in $\Theta(\log N)$ time using only nonatomic reads and writes.* □

3. DEFINITIONS

In this section, we present our model of a nonatomic shared-memory system, which is used in our lower-bound proofs. Our system model is similar to that used in [6, 8].

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.

The class of nonatomic systems we consider is captured by our notion of an event, which is defined below. As we shall see, it is sufficient for our purposes to consider all reads to be atomic, and to allow writes to be either atomic or nonatomic. In our proof, writes to the same variable never overlap each other, so we have no need to define the effects of concurrent writes. According to the definitions below, a read of a variable that overlaps a nonatomic write of that variable may return any value, as assumed earlier in Sec. 2.

An *event* e is denoted $[\mathsf{Op}, R, W, p]$, where $p \in P$. We call Op the *operation* of event e, denoted $op(e)$. Op can be one of the following: $\perp$, $\mathsf{read}(v)$, $\mathsf{write}(v)$, $\mathsf{invoke}(v)$, or $\mathsf{respond}(v)$, where v is a variable in V. (Informally, e can be a local event, an atomic remote read, an atomic remote write, an invocation of a nonatomic remote write, or a corresponding response of a nonatomic remote write. These terms are made precise below.)

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 pair (v, α) in R, and writes the value α to variable v for each pair (v, α) in W. Each variable in R (or W) is assumed to be distinct. We define $Rvar(e)$ ($Wvar(e)$) to be the set of variables read (written) by e, *i.e.*, $Rvar(e) = \{v\text{: } (v, \alpha) \in R\}$ and $Wvar(e) = \{v\text{: } (v, \alpha) \in W\}$. We define $var(e)$, the set of all variables accessed by e, to be $Rvar(e) \cup Wvar(e)$. We say that event e *accesses* each variable in $var(e)$. We say 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 a variable may be remote to *all* processes.) An *initial value* is associated with each variable. An event is *local* if it does not access any remote variables, and *remote* otherwise. The following assumption formalizes requirements stated previously regarding the atomicity of events.

Atomicity Assumption: Each event $e_p = [\mathsf{Op},\ R,\ W,\ p]$ must satisfy one of the conditions below.

- If $\mathsf{Op} = \perp$, then e_p does not access any remote variables. (That is, all variables in $var(e)$ are local to p.) In this case, we call e_p a *local event*.
- If $\mathsf{Op} = \mathsf{read}(v)$, then e_p reads exactly one remote variable, which must be v. In this case, e_p does not write any remote variable and is called a *remote read event*.
- If $\mathsf{Op} = \mathsf{write}(v)$, then e_p writes exactly one remote variable, which must be v. In this case, e_p does not read any remote variable and is called a *remote write event*.
- If $\mathsf{Op} = \mathsf{invoke}(v)$, then $R = \{\}$ and $W = \{(v, \star)\}$, where $\star$ is a special value that means subsequent reads of v may read any value. In this case, e_p is called an *invocation event*.
- If $\mathsf{Op} = \mathsf{respond}(v)$, then e_p writes exactly one remote variable, which must be v. In this case, e_p does not read any remote variable and is called a *response event*. □

Note that an event e_p may write to v if $op(e_p) \in \{\mathsf{write}(v), \mathsf{invoke}(v), \mathsf{respond}(v)\}$, or if v is local to p.

An atomic write is merely a notational convenience to represent a write that is executed "fast enough" to be considered atomic. Therefore, we require that a process has an enabled atomic remote write iff it has an identical enabled nonatomic remote write (P6 below). A nonatomic remote write is represented by two successive events, for its beginning (invocation) and its end (response). If a process has performed an invocation event, then its only enabled event is the matching response (P7 below). As stated before, our proof strategy ensures that between a matching invocation and response, no write to the same variable ever occurs. Thus, as far as overlapping operations are concerned, the only interesting case for us is that of a read of a variable overlapping a write of that same variable. In this case, the read may return any value (P2 and P4 below). For example, in Fig. 2, events a–d may read any value from v.

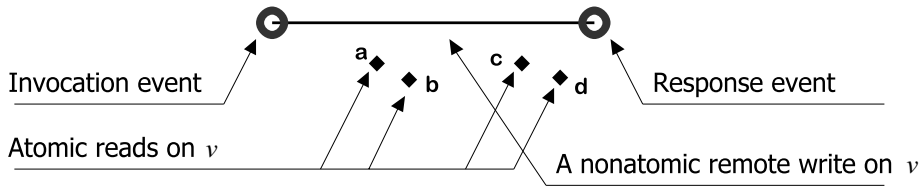

Figure 2: Example of overlapping reads and writes of the same variable.

Note that the Atomicity Assumption allows remote read, remote write, and response events to atomically access any number of local variables, including shared local variables. On the other hand, an invocation event is restricted to access no local variables.

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 $writer\_event(v, H)$,[4] and its owner is denoted $writer(v, H)$. If v is not written by any event in H, then we let $writer(v, H) = \perp$ and $writer\_event(v, H) = \perp$.

We use $\langle e, \ldots \rangle$ to denote a computation that begins with the event e, $\langle\rangle$ to denote the empty computation, and $H \circ G$ to denote the computation obtained by concatenating computations H and G. 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. If G is a subcomputation of H, then $H - G$ is the computation obtained by removing all events in G from H. 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 (*e.g.*, $H \mid p$ instead of $H \mid \{p\}$).

The following properties apply to any shared-memory system.

P1: If $H \in C$ and G is a prefix of H, then $G \in C$. — *Informally, every prefix of a computation is also a computation.*

P2: If $H \circ \langle e_p \rangle \in C$, $G \in C$, $G \mid p = H \mid p$, and if, for all $v \in Rvar(e_p)$, $value(v, G)$ is either $value(v, H)$ or $\star$, then $G \circ \langle e_p \rangle \in C$. — *Informally, if two computations H and G are not distinguishable to process p, if p can execute event e after H, and if all variables in $Rvar(e)$ have the same values (or value $\star$) after H and G, then p can execute e after G.*

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 $op(e_p) = op(e'_p)$. — *Informally, if two computations H and G are not distinguishable to process p, and if p can execute event e after H, then p can execute the same kind of operation after G. (Note that the values read or written might be different.)*

P4: For any $H \in C$, $H \circ \langle e_p \rangle \in C$ implies that either $value(v, H) = \alpha$ or $\star$, for all $(v, \alpha) \in Rvar(e_p)$. — *Informally, only the last value written to a variable may be read, unless the last write was an invocation event on that variable.*

[4]Although our definition of an event allows multiple instances of the same event, we assume that such instances are distinguishable from each other.

P5: For any $H \in C$ and $p \in P$, if $H \circ \langle e_p \rangle \in C$ holds for some invocation event e_p on a variable v, then $H \circ \langle f_p \rangle \in C$ holds for some (atomic) remote write event f_p on v. — *Informally, if p can start writing to v nonatomically (via execution of e_p), then p can also write to v atomically.*

P6: Consider $H \in C$, $e_p = [\mathsf{write}(v),\ R,\ W,\ p]$, $f_p = [\mathsf{invoke}(v), \{\}, \{(v, \star)\}, p]$, and $g_p = [\mathsf{respond}(v),\ R,\ W,\ p]$. Then, $H \circ \langle e_p \rangle \in C$ holds iff $H \circ \langle f_p, g_p \rangle \in C$ holds. — *Informally, p may execute an atomic remote write of v iff it may execute an identical nonatomic write of v.*

P7: For any $H \in C$ and $p \in P$, if e_p and f_p are two consecutive events in $H \mid p$, and if $op(e_p) = \mathsf{invoke}(v)$ for some v, then $op(f_p) = \mathsf{respond}(v)$. — *Informally, an invocation event must be followed by the corresponding response event.*

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 an auxiliary variable $stat_p$ that ranges over $\{ncs,\ entry,\ exit\}$.[5] The variable $stat_p$ is initially ncs and is accessed only by the following events:

$$\begin{aligned} Enter_p &= [\mathsf{write}(stat_p),\ \{\},\ \{(stat_p,\ entry)\},\ p], \\ CS_p &= [\mathsf{write}(stat_p),\ \{\},\ \{(stat_p,\ exit)\},\ p], \text{ and} \\ Exit_p &= [\mathsf{write}(stat_p),\ \{\},\ \{(stat_p,\ ncs)\},\ p]. \end{aligned}$$

The allowable transitions of $stat_p$ are as follows: for all $H \in C$,

$$\begin{aligned} &H \circ \langle Enter_p \rangle \in C \ \text{ iff } \ value(stat_p, H) = ncs; \\ &H \circ \langle CS_p \rangle \in C \ \text{ only if } \ value(stat_p, H) = entry; \\ &H \circ \langle Exit_p \rangle \in C \ \text{ only if } \ value(stat_p, H) = exit. \end{aligned}$$

We define $stat_p$ to be remote to all processes, because this simplifies bookkeeping in our proofs.

We henceforth assume each computation contains at most one $Enter_p$ event for each process p, because this is sufficient for our proof. 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 (livelock freedom): Given $H \in C$, let $X = \{q \in P\colon value(stat_q, H) \neq ncs\}$. If X is nonempty, then there exists an X-computation G such that $H \circ$

[5]Each critical-section execution of p is captured by the single event CS_p, so $stat_p$ changes directly from *entry* to *exit*.

All Proccesses

- $\mathrm{Act}(H)$: Active processes
- $\mathrm{Cvr}(H)$: Covering processes: Covers variables in $V_C(H)$
  - $\mathrm{Cvr}^1(H) = \bigcup_{v \in V_C^1(H)} \mathrm{Cvr}(H, v)$
    Selected at the first induction step; covers variables in $V_C^1(H)$
  - $\mathrm{Cvr}^2(H) = \bigcup_{v \in V_C^2(H)} \mathrm{Cvr}(H, 2)$
    Selected at the second induction step; covers variables in $V_C^2(H)$
  - $\vdots$
  - $\mathrm{Cvr}^m(H) = \bigcup_{v \in V_C^m(H)} \mathrm{Cvr}(H, v)$
    Selected at the m^{th} induction step; covers variables in $V_C^m(H)$
  - For each variable v in $\mathrm{Cvr}^j(H)$,
    $$\mathrm{Cvr}(H, v) = \{cp^j(H, v),\ cp^{j+1}(H, v),\ \ldots,\ cp^m(H, v)\} \cup \mathrm{Res}(H, v),$$
    where $\mathrm{Res}(H, v)$ is the set of reserve processes for v
- Erased processes: Perform no events in H

Figure 3: Process groups for a regular computation H. In this figure, m is the induction number of H.

$G \circ \langle e_p \rangle \in C$, where $p \in X$ and e_p is either CS_p (if $value(stat_p, H) = entry$) or $Exit_p$ (if $value(stat_p, H) = exit$).

4. LOWER BOUNDS

The proof of our $\Omega(\log N / \log \log N)$ lower bound focuses on a special class of computations called "regular" computations. A regular computation consists of events of two groups of processes, "active processes" and "covering processes." Informally, an active process is a process in its entry section, competing with other active processes; a covering process is a process that has executed some part of its entry section, and executes (or is ready to execute) a nonatomic write (*i.e.*, an invocation event followed by a corresponding response event) on some variable v in order to "cover" v, so that other processes may concurrently access v without gathering knowledge of each other.

In our proof, longer and longer regular computations are inductively constructed until the lower bound is attained. Information flow among processes is prevented either by covering variables, as described above, or by erasing processes — when a process is erased, its events are completely removed from the computation currently being considered. These basic techniques, covering and erasing, have been previously used to prove other lower bounds pertaining to concurrent systems [2, 6, 11, 17, 28]. However, the particular covering strategy being used here is different from those applied in earlier papers, as it strongly exploits the fact that nonatomic writes may occur for arbitrary durations.

Covering is a useful technique because erasing alone sometimes does not leave enough active processes for the next induction step. This happens only if some variables are written by "sufficiently many" of the remote events being appended to the current computation. In this case, for each such variable v, some active processes are changed to covering processes. These covering processes execute their next remote (write) event nonatomically, in order to prevent information flow via future accesses to v. We guarantee that any subsequent read from v happens concurrently with a nonatomic write to v (by some covering process). By our system model, any process that subsequently reads v may read any value. Thus, information flow can be prevented by assuming that each such process p reads the initial value of v (if p has not written to v before) or the value p last wrote to v (otherwise).

Having outlined some of the basic ideas of our proof, we now define some relevant notation and terminology. The last of these definitions is the notion of a regular computation, mentioned above. After formally defining the class of regular computations, we give a detailed proof sketch. As mentioned earlier, a detailed proof is given in an appendix.

A regular computation H has an associated *induction number* m, which is the number of induction steps taken to construct H. Such a computation H can be written $H = H^1 \circ H^2 \circ \cdots \circ H^m$, where H^j is called the j^{th} *segment* of H. H^j includes the events appended at the j^{th} induction step. We now explain the process groups involved in constructing H in detail. These processes groups are depicted in Fig. 3.

The processes participating in H are partitioned into two sets: $\mathrm{Act}(H)$, the *active processes*, and $\mathrm{Cvr}(H)$, the *covering processes*. The set of variables covered by $\mathrm{Cvr}(H)$ is denoted $V_C(H)$. The covering processes may be grouped in two ways: $\mathrm{Cvr}(H) = \bigcup_{j=1}^{m} \mathrm{Cvr}^j(H)$, where $\mathrm{Cvr}^j(H)$ is the set of covering processes added to $\mathrm{Cvr}(H)$ at the j^{th} induction step; or $\mathrm{Cvr}(H) = \bigcup_{v \in V_C(H)} \mathrm{Cvr}(H, v)$, where $\mathrm{Cvr}(H, v)$ is the set of processes that cover variable v.

If a variable v is chosen to be covered at the j^{th} induction step, then we define $\mathrm{Cvr}(H, v)$ to be large enough so that v can be covered throughout the rest of the induction. Therefore, each v is selected to be covered at most once, and hence we can partition $V_C(H)$ into a disjoint union $\bigcup_{j=1}^{m} V_C^j(H)$, where $V_C^j(H)$ is the set of variables selected to be covered at the j^{th} induction step. Each set $\mathrm{Cvr}^j(H)$ can be also written as a disjoint union $\bigcup_{v \in V_C^j(H)} \mathrm{Cvr}(H, v)$.

We now explain more closely the structure of $\mathrm{Cvr}(H, v)$, for a given variable v. Assume that $v \in V_C^j(H)$, *i.e.*, v and $\mathrm{Cvr}(H, v)$ are selected at the j^{th} induction step. Then, the

processes in $\mathrm{Cvr}(H, v)$ execute together with active processes prior to the j^{th} induction step. At the j^{th} step, $\mathrm{Cvr}(H, v)$ is constructed. At most one process in $\mathrm{Cvr}(H, v)$ is actually needed in step j to cover v, so at most one such process is selected to cover v *at the* j^{th} *step*. We denote this process as $cp^j(H, v)$. (If no process is selected, then we define $cp^j(H, v) = \perp$.) All events that access v in segment H^j are executed after the invocation event on v by $cp^j(H, v)$ (if not $\perp$), and before the corresponding response event.

At each successive induction step, at most one process among $\mathrm{Cvr}(H, v)$ is selected to cover v. This is done to ensure that each nonatomic write eventually terminates, and that writes of the same variable do not overlap. Thus, the set $\mathrm{Cvr}(H, v)$ includes each $cp^k(H, v)$, where $k = j, j+1, \ldots, m$ and $cp^k(H, v) \neq \perp$, plus perhaps some additional processes. We define $\mathrm{Res}(H, v)$, the set of *reserve processes for* v *in* H, to be the set of these additional processes, *i.e.*, $\mathrm{Res}(H, v) = \{p\text{: } p \in \mathrm{Cvr}(H, v) \text{ and } p \neq cp^k(H, v) \text{ for all } k \leq m\}$. (For notational convenience, we define $cp^k(H, v) = \perp$ for $k < j$.) The processes in $\mathrm{Res}(H, v)$ serve two purposes: they are used to cover v in further induction steps (*i.e.*, beyond the m^{th}), and in case some process $cp^k(H, v)$ has to be erased in future induction steps, a reserve process is selected to take its place (*i.e.*, to become the new $cp^k(H_{\text{new}}, v)$, where H_{new} is the future computation being considered).

H^j, which is appended to $H^1 \circ H^2 \circ \cdots \circ H^{j-1}$ at the j^{th} induction step, consists of four subsequences of events. The first is the *local segment* L^j, which consists of local events of processes that are active in H^{j-1}. These events cannot create any information flow. (L^j consists of events of processes in $\mathrm{Act}(H)$ as well as $\mathrm{Cvr}^k(H)$, for $k \geq j$. Recall that a process in $\mathrm{Cvr}^k(H)$ does not become a covering process until the k^{th} induction step.) After adding the local segment, we select the variables $V_C^j(H)$ to be covered and the processes $\mathrm{Cvr}^j(H)$ that will be used to cover those variables. (The selection method is explained in detail below.) After that, we consider each v, where $v \in V_C^k(H)$ for some $k \leq j$, *i.e.*, v is a variable that is selected for covering by the j^{th} step. For each such v, we choose a process in $\mathrm{Cvr}(H, v)$ as $cp^j(H, v)$, *i.e.*, as the process that covers v in the j^{th} induction step. The *invocation segment* I^j is either empty (if no covering is required for this induction step), or consists of one invocation event for each chosen $cp^j(H, v)$. Next, we construct the *remote segment* M^j, in which non-covering processes execute remote events. (As in the case of the local segment, M^j consists of events of processes in $\mathrm{Act}(H)$ as well as $\mathrm{Cvr}^k(H)$, for $k > j$. Note that $\mathrm{Cvr}^j(H)$ is now excluded.) Finally, the *response segment* R^j consists of matching response events for each invocation event in I^j. The structure of a regular computation, explained so far, is depicted in Fig. 4.

We now formally define the notion of a regular computation.

Definition: Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system. A computation H in C is *regular* iff it satisfies the following.

The computation H can be written $H = H^1 \circ H^2 \circ \cdots \circ H^m$, where H^j is called the j^{th} *segment* of H. We call m the *induction number* of H. H^j includes the events appended at the j^{th} induction step, and can be subdivided as $H^j = L^j \circ I^j \circ M^j \circ R^j$, as explained below.

- The *local segment* L^j consists of local events by processes in $\mathrm{Act}(H)$ and $\bigcup_{k=j}^{m} \mathrm{Cvr}^k(H)$.
- The *invocation segment* I^j is either $\langle\rangle$, or consists of one invocation event on v by $cp^j(H, v)$, for each $v \in \bigcup_{k=1}^{j} V_C^k(H)$.
- The *remote segment* M^j consists of local, remote read, and/or remote write (but not invocation or response) events by processes in $\mathrm{Act}(H)$ and $\bigcup_{k=j+1}^{m} \mathrm{Cvr}^k(H)$.
- The *response segment* R^j consists of response events that correspond to events in I^j.

Moreover, H satisfies the following regularity conditions.

RC1: Assume that H can be written as $E \circ \langle e_p \rangle \circ F \circ \langle f_q \rangle \circ G$. If $p \neq q$ and there exists a variable $v \in \mathit{Wvar}(e_p) \cap \mathit{Rvar}(f_q)$ such that F does not contain a write to v (*i.e.*, $\mathit{writer\_event}(v, F) = \perp$), then $p \in \mathrm{Cvr}(H, v)$ and e_p is an invocation event on v. — *Informally, if p is the last process to write variable v, and if another process q subsequently reads v, then p covers v, and hence q can read any value.*

RC2: For any event e_p in H and any variable v in $\mathit{var}(e_p)$, if v is local to another process q ($\neq p$), then $H \mid q = \langle\rangle$. — *Informally, if no process accesses a local variable of other participating processes.*

RC3: If there exists two distinct processes p and q that both write to v in H, then $v \in V_C(H)$. — *Informally, any variable that is written by multiple processes are covered.*

RC4: For any process p such that $H \mid p \neq \langle\rangle$, $\mathit{value}(\mathit{stat}_p, H) = \mathit{entry}$ holds. — *Informally, every participating process is in its entry section.*

RC5: For any process p such that $H \mid p \neq \langle\rangle$, $H \mid p \in C$ holds. Moreover, if $p \in \mathrm{Res}(H, v)$ for some v, then $(H \mid p) \circ \langle e_p, f_p \rangle \in C$ holds, where e_p (f_p) is some invocation (response) event on v. — *Informally, every process executes the same computation as its solo computation. Moreover, if p is a reserve process for v, then it is ready to write to v as its "next" event.* □

Detailed proof overview. Initially, we start with H_1, in which $\mathrm{Act}(H_1) = P$, $\mathrm{Cvr}(H_1) = \{\}$, and each process has one remote event. At the j^{th} induction step, we consider a computation $H_j = H^1 \circ H^2 \circ \cdots \circ H^j$ such that $\mathrm{Act}(H_j)$ consists of n processes, each of which executes j remote events. We show that if $j > \log n$,[6] then the lower bound has already been attained. Thus, in the inductive step, we assume $j \leq \log n$. Moreover, we also assume that for each covered variable v, its set of covering processes $\mathrm{Cvr}(H, v)$ has exactly c elements, where $c = \Theta(\log n)$. We construct a regular computation $H_{j+1} = H_j \circ H^{j+1}$ with induction number $j + 1$ such that $\mathrm{Act}(H_{j+1})$ consists of $\Omega(n/\log^2 n)$ processes, each of which executes $j + 1$ remote events. The construction method, formally described in Lemma 4 in the appendix, is explained below.

We show in Lemma 3 that, among the n processes in $\mathrm{Act}(H_j)$, at least $n - 1$ processes can execute an additional remote event before entering its critical section. We call

[6] We use $\log n$ to denote $\log_2 n$ (base-2 logarithm), and use $\log^2 n$ to denote $(\log_2 n)^2$.

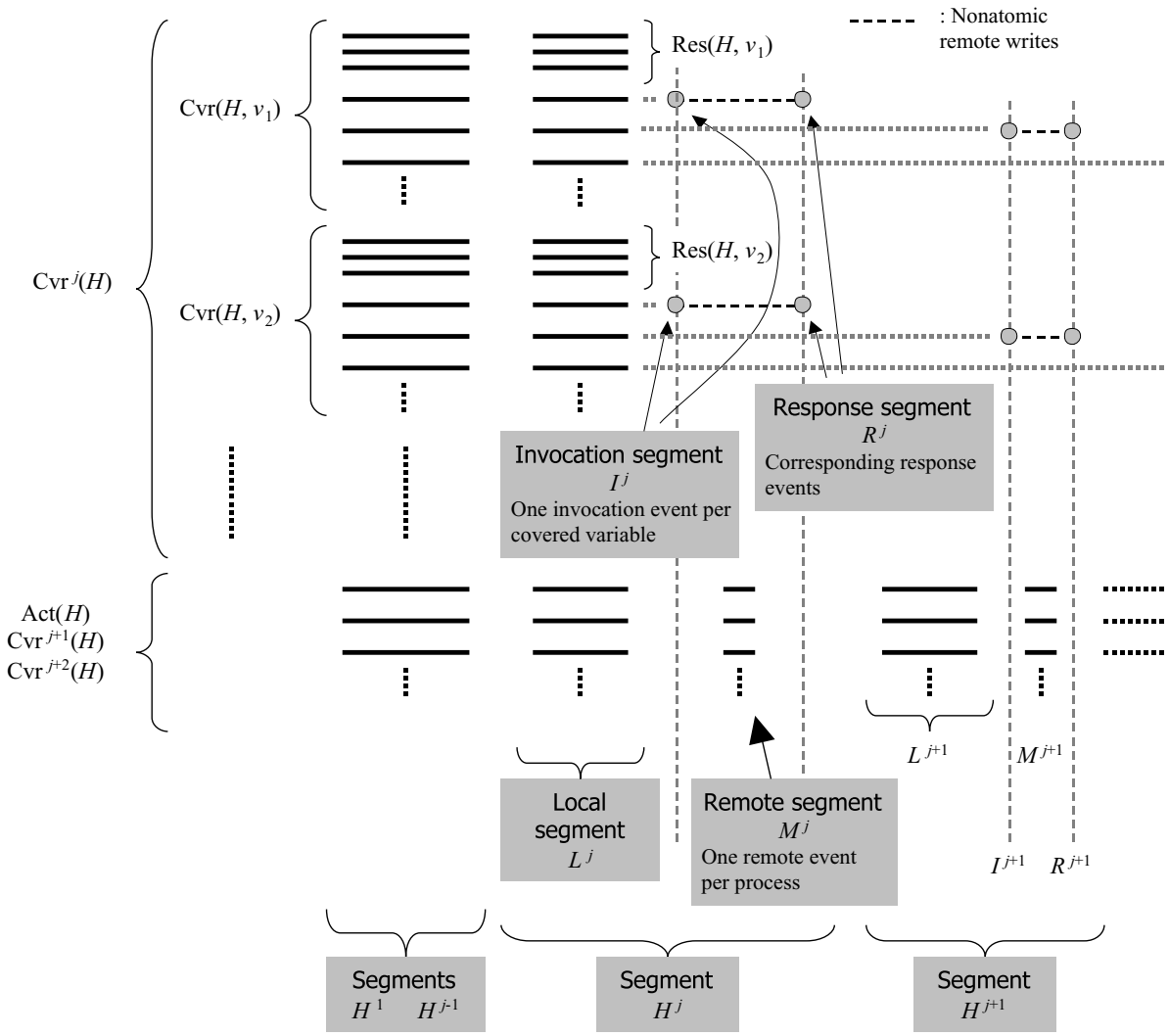

Figure 4: The structure of a regular computation. The computation is depicted as a collection of bold horizontal lines, with each line representing the events of a single process. In addition, filled circles are used to denote the invocation and response events of nonatomic writes by covering processes. Only one segment H^j is shown in detail. In this figure, $v_1,\ v_2,\ \ldots$ are variables in $V_C^j(H)$. For simplicity, processes in $\mathrm{Cvr}^1(H)$, $\mathrm{Cvr}^2(H),\ \ldots, \mathrm{Cvr}^{j-1}(H)$ are not shown, and those in $\mathrm{Cvr}^{j+1}(H),\ \mathrm{Cvr}^{j+2}(H),\ \ldots$ are not shown in detail. These sets include covering processes for variables selected for covering at induction steps before and after step j. The manner in which they execute is similar to that depicted for $\mathrm{Cvr}^j(H)$. As shown in the figure, some processes in $\mathrm{Cvr}^j(H)$ are selected to cover their respective variables at both induction steps j and $j+1$ (and possibly future steps as well). In general, covering may or may not be needed at a given induction step.

these events "next" remote events, and denote the corresponding set of processes by Y. First, we eliminate conflicts *between* Y and $\mathrm{Cvr}(H)$, as described next. (A process conflicts with another if information flow is possible, or a regularity condition is violated.)

Each next event by processes in Y may conflict with a covering process. In particular, consider a process $p \in Y$ that accesses a remote variable v, and a process $q \in \mathrm{Cvr}(H)$. If v is local to q, then RC2 is violated. (Note that, if RC2 is violated, then a *local* event of q may generate information flow, which is clearly undesirable.) Otherwise, if q has (atomically) written to v in H, and if v is not yet covered, then a conflict arises, and hence either p or q must be erased. (Formally, if p reads v, then RC1 is violated; if p overwrites v, then RC3 is violated.)

By RC3, it can be shown that each process in Y may conflict with at most one process in $\mathrm{Cvr}(H)$. Since each variable is covered by $c = \Theta(\log n)$ processes, we can partition $\mathrm{Cvr}(H)$ into c subsets $X_1,\ X_2,\ \ldots,\ X_c$ such that each subset contains exactly one covering process for each v, as depicted in Fig. 5(a). Then, we can similarly partition Y into $Y_1,\ Y_2,\ \ldots,\ Y_c$, such that a process in Y_k conflicts only with a process in X_k.

We choose the largest subset Y_k, and erase the rest of Y. It is clear that $|Y_k| \geq Y/c = \Theta(n/\log n)$. By also erasing X_k, we guarantee that there is no conflict between Y_k and

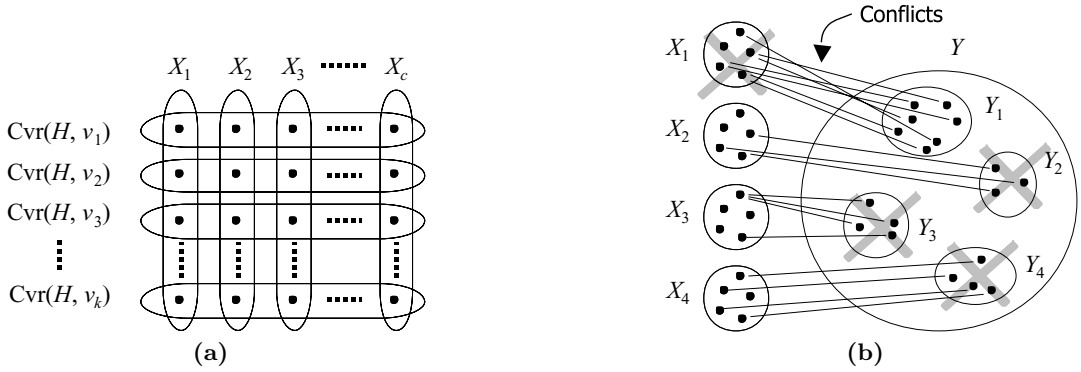

Figure 5: (a) Partition of $\mathrm{Cvr}(H)$. In this figure, we define $V_C(H)$ as $\{v_1, v_2, \ldots, v_k\}$. Each dot represents a process in $\mathrm{Cvr}(H)$. Note that, by assumption, $|\mathrm{Cvr}(H, v_j)| = c$ holds for each $v_j \in V_C(H)$. (b) Elimination of conflicts when $c = 4$ and $V_C(H)$ contains 5 variables. Note that in Y, only the largest subset, Y_1, survives.

the covering processes. This is illustrated in Fig. 5(b). (By RC1, the reserve processes in X_k may be erased without any difficulty. If a process q in X_k is *not* a reserve process, then we can exchange q with a reserve process for the same covering variable, and then erase q.)

The set of processes Y_k can be divided into two subsets (one of which may be empty): the set of *readers*, which have a next remote event that is a remote read, and the set of *writers*, which have a next remote event that is a remote write. We define Y to be the larger of the two, and erase the smaller subset. If the writers are saved, then no covering is necessary for this induction step, since in this case no next remote event reads a remote variable. Therefore, I^{j+1} and R^{j+1} are nonempty iff the readers are saved.

The processes in Y collectively execute $\Theta(n/\log n)$ next remote events. Among the variables that are accessed by these events *and* are not covered yet, we identify V_{HC}, the set of "high contention" variables that are remotely accessed by at least $d \log n$ next remote events, where d is a constant to be specified. Then, we have $|V_{\mathrm{HC}}| \leq |Y|/(d \log n) = \Theta(n/\log^2 n)$. Next, we partition the processes in Y, depending on whether their next remote events access a variable in V_{HC}, as follows: $P_{\mathrm{HC}} = \{y \in Y$: y's next remote event accesses some variable in $V_{\mathrm{HC}}\}$, and $P_{\mathrm{LC}} = Y - P_{\mathrm{HC}}$.

Because H_j is regular and $Y \subseteq \mathrm{Act}(H_j)$, we can erase any process in Y. Hence, we can erase the smaller of P_{HC} and P_{LC} and still have $\Theta(n/\log n)$ remaining active processes. We consider these cases separately.

Erasing strategy. Assume that we erased P_{HC} and saved P_{LC}. This situation is depicted in Fig. 6. Define V_{LC} as the set of variables remotely accessed by the next remote events of P_{LC}. Then clearly, every non-covered variable in V_{LC} is a "low contention" variable, and hence is accessed by at most $d \log n$ different next remote events (and hence, different processes). Therefore, a next remote event by a process in P_{LC} can conflict with at most $d \log n$ processes. By generating a "conflict graph" and applying Turán's theorem (Theorem 4), we can find a set of processes Z such that $|Z| = \Omega(n/\log^2 n)$ and their remote events do not conflict with each other. By retaining Z and erasing all other active processes, we can eliminate all conflicts and thereby construct H_{j+1}. (Note that no new covering variables/processes are introduced in this strategy.)

Covering strategy. Assume that we erased P_{LC} and saved P_{HC}. Every next remote event by a process in P_{HC} accesses a variable in V_{HC}. If all next remote events are remote reads (*i.e.*, we saved the readers above), then we can eliminate all conflicts by erasing at most one process per each variable. (This is because, by RC3, each non-covered variable is written by at most one process.) Thus, assume that the next remote events consist entirely of remote writes (*i.e.*, we saved the writers above). For each variable $v \in V_{\mathrm{HC}}$, there exists at least $d \log n$ processes in P_{HC} that remotely write to v. Thus, we can choose $\Theta(\log n)$ processes for each $v \in V_{\mathrm{HC}}$ to be the new covering processes, and still have $\Theta(|P_{\mathrm{HC}}|) = \Theta(n/\log n)$ remaining active processes. The set V_{HC} becomes $V_C^{j+1}(H_{j+1})$ for the next induction step, and for each $v \in V_{\mathrm{HC}}$, $\Theta(\log n)$ processes become $\mathrm{Cvr}(H_{j+1}, v)$. Thus, we can construct H_{j+1}.

A detailed version of the proof sketched above is given in the appendix. From this proof, we have the following.

THEOREM 2. *For any mutual exclusion system $\mathcal{S} = (C, P, V)$, there exist a p-computation F such that F does not contain CS_p, and p executes $\Omega(\log N/\log\log N)$ remote events in F, where $N = |P|$.*

This theorem can be adapted to apply to systems in which caches are used to avoid accessing the interconnection network, by using a technique of Anderson and Yang [8] to count the number of distinct remote variables accessed by each process. In this technique, certain remote events are classified as "critical." An event is *critical* if it is an expanding read, an expanding write, or a nonexpanding critical event. An *expanding read* (resp. *expanding write*) of a remote variable v by a process p is the first event by p that reads (resp. writes) v. A *nonexpanding critical event* is an event of p that accesses a variable v for the first time after some expanding read or write of p.

Suppose that we have a regular computation H_j, and a "next" remote event e_p of a process $p \in \mathrm{Act}(H_j)$. If e_p is not a critical event, then we can rearrange the events of p, including e_p, with respect to events of other processes, such that e_p does not generate any conflicts. In that way, the only events that may possibly generate conflicts are critical events; noncritical events can be appended without any erasing or covering, just like local events.

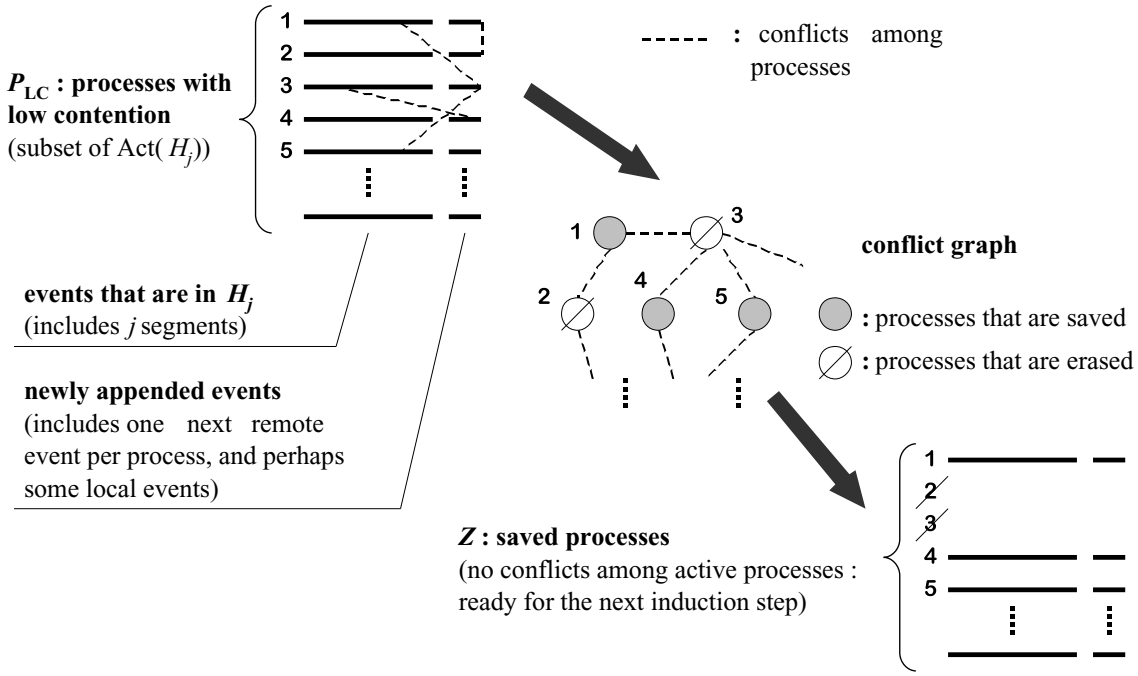

Figure 6: Erasing strategy. For simplicity, covering processes are not shown in this figure.

Unfortunately, since events are reordered in the middle of the induction, concurrent writes of the same variable inevitably result. Therefore, this proof strategy only works if we change Properties P2 and P4 as follows:

P2′: Assume that $H \circ \langle e_p \rangle \in C$, $G \in C$, and $G \mid p = H \mid p$. Also assume that, for all $v \in Rvar(e_p)$, either $value(v, G) = value(v, H)$ holds, or there exists a process q such that the last event of $G \mid q$ has operation $\mathsf{invoke}(v)$. Then, $G \circ \langle e_p \rangle \in C$. — *Informally, if two computations H and G are not distinguishable to process p, if p can execute event e after H, and if for each variable v in $Rvar(e)$, either v has the same value after H and G or some process q is concurrently executing a nonatomic write of v, then p can execute e after G.*

P4′: For any $H \in C$, $H \circ \langle e_p \rangle \in C$ implies that, for all $(v, \alpha) \in Rvar(e_p)$, either $value(v, H) = \alpha$ holds or there exists a process q such that the last event of H in q is an invocation event on v. — *Informally, only the last value written to a variable may be read, unless there exists a concurrent nonatomic write of that variable.*

Thus, we can show that there exists a solo p-computation such that p executes $\Omega(\log N/\log\log N)$ *critical* events in H. Moreover, we can also show that, if p accesses D distinct remote variables, then at most $O(D)$ nonexpanding critical events may occur between two successive expanding reads and/or writes. Note that, since each expanding read (resp. write) accesses a different remote variable, the number of expanding reads (resp. writes) is at most D. Since we have $O(D)$ nonexpanding critical events between each consecutive pair of expanding critical events, the total number of critical events is $O(D^2)$. In other words, we have $D = \Omega(\sqrt{m})$, where m is the number of critical events, that is, the induction number of the final computation. Thus, we have the following theorem.

THEOREM 3. *For any mutual exclusion system $\mathcal{S} = (C, P, V)$ that satisfies Properties P2′ and P4′ instead of P2 and P4, there exist a p-computation F such that F does not contain CS_p, and p accesses $\Omega(\sqrt{\log N/\log\log N})$ distinct remote variables in F, where $N = |P|$.*

5. CONCLUDING REMARKS

We have presented a nonatomic local-spin mutual exclusion algorithm with $\Theta(\log N)$ worst-case time complexity, which matches that of the best atomic algorithm proposed to date. We have also shown that for any N-process nonatomic algorithm, there exists a single-process execution in which the lone competing process performs $\Omega(\log N/\log\log N)$ remote memory references that access $\Omega(\sqrt{\log N/\log\log N})$ distinct variables in order to enter its critical section. These bounds show that fast and adaptive algorithms are impossible if variable accesses are nonatomic, even if caching techniques are used to avoid accessing the processors-to-memory interconnection network.

Our work suggests several avenues for further research. The most obvious is to close the gap between our $\Theta(\log N)$ algorithm and our $\Omega(\log N/\log\log N)$ and $\Omega(\sqrt{\log N/\log\log N})$ lower bounds. We conjecture that $\Omega(\log N)$ is a tight lower bound on the number of distinct variables remotely accessed, even when restricting attention to single-process executions. Another interesting question arises from our lower-bound proofs. These proofs hinge on the ability to "stall" nonatomic writes for arbitrarily long intervals. This gives rise to the following question: Is it possible to devise a nonatomic algorithm that is fast or adaptive if each write is guaranteed to complete within some bound Δ? We hope to resolve this question in future work.

6. 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. ACM, 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. ACM, July 2000.

[3] R. Alur and G. Taubenfeld. Contention-free complexity of shared memory algorithms. *Information and Computation*, 126(1):62–73, April 1996.

[4] J. Anderson. A fine-grained solution to the mutual exclusion problem. *Acta Informatica*, 30(3):249–265, May 1993.

[5] 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.

[6] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 90–99, August 2001.

[7] J. Anderson and Y.-J. Kim. A new fast-path mechanism for mutual exclusion. *Distributed Computing*, 14(1):17–29, January 2001.

[8] J. Anderson and J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.

[9] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[10] 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. ACM, July 2000.

[11] 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.

[12] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

[13] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, June 1990.

[14] S. Haldar and P. Subramanian. Space-optimum conflict-free construction of 1-writer 1-reader multivalued atomic variable. In *Proceedings of the Eighth International Workshop on Distributed Algorithms*, pages 116–129. Lecture Notes in Computer Science 857, Springer-Verlag, 1994.

[15] S. Haldar and K. Vidyasakar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *Journal of the ACM*, 42(1):186–203, 1995.

[16] J. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17:135–141, 1982.

[17] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 1–15, October 2001.

[18] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[19] L. Lamport. The mutual exclusion problem: Part II - Statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.

[20] L. Lamport. On interprocess communication: Part II - Algorithms. *Distributed Computing*, 1:86–101, 1986.

[21] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[22] L. Lamport. win and sin: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12(3):396–428, July 1990.

[23] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[24] G. Peterson and J. Burns. Concurrent reading while writing II: The multi-writer case. In *Proceedings of the 28th Annual ACM Symposium on Foundation of Computer Science*. ACM, 1987.

[25] R. Schaffer. On the correctness of atomic multi-writer registers. Technical Report MIT/LCS/TM-364, Laboratory for Computer Science, MIT, Cambridge, 1988.

[26] A. Singh, J. Anderson, and M. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):311–339, 1994.

[27] E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 159–168. ACM, August 1992.

[28] 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. ACM, August 1989.

[29] P. Turán. On an extremal problem in graph theory (in Hungarian). *Mat. Fiz. Lapok*, 48:436–452, 1941.

[30] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.

Appendix: Detailed Proofs of Theorems 2 and 3

In this appendix, we present detailed proofs of Theorems 2 and 3. Throughout this appendix, we assume the existence of a fixed mutual exclusion system $\mathcal{S} = (C, P, V)$. We begin by defining two "operators" on regular computations.

Definition: Consider a regular computation H in C and a process p. If $p \in \mathrm{Act}(H)$, or if $p \in \mathrm{Res}(H, v)$ for some variable v, then $erase_p(H)$ is defined to be $H - (H \mid p)$. □

Definition: Consider a regular computation H in C and two processes p and q. Assume that $p \in \mathrm{Cvr}(H, v) - \mathrm{Res}(H, v)$ and $q \in \mathrm{Res}(H, v)$ for some variable v. We define the exchange operator $exchange_{pq}$ as follows.

By the definition of $\mathrm{Cvr}(H, v)$ and $\mathrm{Res}(H, v)$, we have $p = cp^j(H, v)$, for some j. Thus, I^j, the j^{th} invocation segment of H, contains an invocation event e_p on v. Similarly, R^j, the j^{th} response segment of H, contains the corresponding response event f_p. It follows that H^j can be written as $H^j = E \circ \langle e_p \rangle \circ F \circ \langle f_p \rangle \circ G$. Also, since $q \in \mathrm{Res}(H, v)$, by RC5, there exist an invocation event e_q and a response event f_q on v such that $(H \mid q) \circ \langle e_q, f_q \rangle \in C$.

We define $exchange_{pq}(H)$ to be the computation obtained by replacing e_p and f_p by e_q and f_q, *i.e.*,

$$\begin{aligned} exchange_{pq}(H) \;=\; & H^1 \circ H^2 \circ \cdots \circ H^{j-1} \circ \\ & E \circ \langle e_q \rangle \circ F \circ \langle f_q \rangle \circ G \circ \\ & H^{j+1} \circ H^{j+1} \circ \cdots \circ H^m, \end{aligned}$$

where m is the induction number of H. □

Informally, $exchange_{pq}$ is an operator that exchanges the role of a non-reserve process p and a reserve process q, if both cover the same variable v. We claim that these two operators preserve regularity and the structure of H (*e.g.*, $\mathrm{Act}(H)$, $\mathrm{Cvr}(H)$, *etc.*). This claim is formalized in the following two lemmas.

Lemma 1 *Consider a regular computation H in C with induction number m, and a process p. Assume that either $p \in \mathrm{Act}(H)$ or $\{p\} \subsetneq \mathrm{Res}(H, v)$ holds for some v, and define $H' = erase_p(H)$. Then H' is a regular computation in C with induction number m, satisfying the following, for each $u \in V_C(H)$ and each j $(1 \le j \le m)$:*

$$\begin{aligned} \mathrm{Act}(H') &= AP(H) - \{p\}; \\ \mathrm{Cvr}(H') &= \mathrm{Cvr}(H) - \{p\}; \\ V_C(H') &= V_C(H); \\ \mathrm{Cvr}(H', u) &= \begin{cases} \mathrm{Cvr}(H, v) - \{p\}, & \text{if } u = v \\ \mathrm{Cvr}(H, u), & \text{otherwise;} \end{cases} \\ cp^j(H', u) &= cp^j(H, u). \end{aligned}$$ □

Lemma 2 *Consider a regular computation H in C with induction number m, and two process p and q. Assume that $p \in \mathrm{Cvr}(H, v) - \mathrm{Res}(H, v)$ and $q \in \mathrm{Res}(H, v)$ for some v, and define $H' = exchange_{pq}(H)$. Then H' is a regular computation in C with induction number m, satisfying the following, for each $u \in V_C(H)$ and each j $(1 \le j \le m)$:*

$$\begin{aligned} \mathrm{Act}(H') &= AP(H); \\ \mathrm{Cvr}(H') &= \mathrm{Cvr}(H); \\ V_C(H') &= V_C(H); \\ \mathrm{Cvr}(H', u) &= \mathrm{Cvr}(H, u); \\ cp^j(H', u) &= \begin{cases} q, & \text{if } cp^j(H, u) = p \\ cp^j(H, u), & \text{otherwise.} \end{cases} \end{aligned}$$ □

The next lemma states that if n active processes are competing for entry into their critical sections, then at least $n-1$ of them must execute at least one more remote event before entering their critical sections.

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 $n - 1 \le |Y| \le n$, satisfying the following: for each process p in Y, there exist a local p-computation L_p, a remote event e_p of p, and a variable v_p such that $(H \mid p) \circ L_p \circ \langle e_p \rangle \in C$, $e_p \neq CS_p$, and $op(e_p)$ is either $\mathsf{write}(v_p)$ or $\mathsf{read}(v_p)$.* □

The following theorem is due to Turán [29].

THEOREM 4 (TURÁN). *Let $\mathcal{G} = (V, E)$ be an undirected graph with vertex set V and edge set E. If the average degree of $\mathcal{G}$ is d, then an independent set[7] exists with at least $\lceil |V|/(d+1) \rceil$ vertices.*

The following lemma provides the induction step that leads to the lower bound in Theorem 2.

Lemma 4 *Let H be a regular computation in C with induction number m. Define $n = |\mathrm{Act}(H)|$. Assume that $n \ge 8$ and*

- $m \le \log n - 2$; (1)
- *each process in $\mathrm{Act}(H)$ executes exactly one remote event in each remote segment M^j, for $j = 1, \ldots, m$;* (2)
- $|\mathrm{Cvr}(H, v)| = c$ *holds for each $v \in V_C(H)$, where $c \ge 2 \log n$.* (3)

Then, there exists a regular computation G in C with induction number $m + 1$, such that

- $\mathrm{Act}(G) \subseteq \mathrm{Act}(H)$;
- $n' \ge \min\left\{ \dfrac{(n-1)(2\log n - 5)}{24(2\log n + 1)^2}, \dfrac{n-1}{4(8\log n + 3)^2} \right\}$, *where $n' = |\mathrm{Act}(G)|$;* (4)
- *each process in $\mathrm{Act}(G)$ executes exactly one remote event in each remote segment M^j, for $j = 1, \ldots, m+1$;* (5)
- $|\mathrm{Cvr}(H, v)| = c'$ *holds for each $v \in V_C(H')$, where $c' \ge 2 \log n'$.* (6)

Proof: By the definition of $V_C(H)$, for each variable $v \in V_C(H)$,

$$|\{q : q \neq \bot \text{ and } q = cp^j(H, v) \text{ for some } j = 1, 2, \ldots, m\}| \le m.$$

Thus, by (3),

$$|\mathrm{Res}(H, v)| \ge c - m, \text{ for each } v \in V_C(H). \tag{7}$$

If $c > \lceil 2 \log n \rceil$, then by (1), we have $c > m$. Therefore, for each $v \in V_C(H)$, $\mathrm{Res}(H, v)$ is nonempty. In this case, we can choose one reserve process q_v from each $\mathrm{Res}(H, v)$, and by inductively applying $erase_{q_v}$, and using Lemma 1, obtain another computation $\overline{H}$ that satisfies assumptions (1)–(3) with 'c' $\leftarrow c - 1$.

[7] 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'.

We can repeat this procedure until $c = \lceil 2 \log n \rceil$ is established. Therefore, in order to prove the lemma, it suffices to consider the case when

$$c = \lceil 2 \log n \rceil. \tag{8}$$

We assume this throughout the remainder of the proof. Therefore, by (1), we have $c - m \geq 2$. Thus, by (7),

- $|\mathrm{Res}(H, v)| \geq 2$, for each $v \in V_C(H)$. (9)

Because H is regular, by Lemma 3, there exists a subset Y of $\mathrm{Act}(H)$ such that

$$n - 1 \leq |Y| \leq n, \tag{10}$$

and for each process $p \in Y$, there exist a local p-computation L_p, a "next" remote event e_p of p, and a variable v_p such that,

- $(H \mid p) \circ L_p \circ \langle e_p \rangle \in C$, (11)
- $e_p \neq CS_p$, and
- $op(e_p)$ is either $\mathsf{read}(v_p)$ or $\mathsf{write}(v_p)$. (12)

We now eliminate any conflicts between Y and $\mathrm{Cvr}(H)$. By (3), we can partition $\mathrm{Cvr}(H)$ into disjoint sets X_1, X_2, $\ldots$, X_c of equal size, such that each X_j contains exactly one process in each $\mathrm{Cvr}(H, v)$. We now partition Y into Y_1, Y_2, $\ldots$, Y_c, such that each process in Y_j only conflicts with processes in X_j, as follows.

For each $p \in Y$, we apply the following rules: **(R1)** if v_p is local to a process $q \in X_j$, then put p into Y_j; **(R2)** if $v_p \notin V_C(H)$ and there exists a process $r \in X_j$ that writes to v_p in H, then put p into Y_j; **(R3)** otherwise, we can put p into any Y_j, say, Y_1.

Note that, if Rules R1 and R2 are both applicable, then by RC2, we have $q = r$. Also, by RC3, Rule R2 is applicable to at most one process r. Thus, Rules R1–R3 generate a well-defined partition of Y into c distinct subsets. Choose $j_{\max}$ such that $Y_{j_{\max}}$ is the largest subset among Y_1, $\ldots$, Y_c. Then, by (10),

$$|Y_{j_{\max}}| \geq (n-1)/c. \tag{13}$$

By (12), for each $p \in Y_{j_{\max}}$, e_p is either a remote read or a remote write event. Hence, we can find a subset Y' of $Y_{j_{\max}}$ such that

- $|Y'| = \lceil (n-1)/2c \rceil$, and (14)
- either *all* events e_p (for $p \in Y'$) are remote reads, or all are remote writes. (15)

We now construct a computation H' by erasing all processes in $\mathrm{Act}(H) - Y'$ and in $X_{j_{\max}}$. The construction algorithm is depicted in Fig. 7, and explained below.

By inductively applying $erase_p$ for each $p \in \mathrm{Act}(H) - Y'$, and using Lemma 1, we can erase all processes in $\mathrm{Act}(H) - Y'$. Also, we can erase all processes in $X_{j_{\max}}$ as follows. If a process $q \in X_{j_{\max}}$ is a reserve process, then apply $erase_q$ and Lemma 1. Otherwise, we have $q \in \mathrm{Cvr}(H, v) - \mathrm{Res}(H, v)$ for some v, and hence $q = cp^k(H, v)$ for some v and k. By (9), there exists a process $r \in \mathrm{Res}(H, v)$. Since q and r cover the same variable v, from the construction of X_1, $\ldots$, X_c, we have $r \notin X_{j_{\max}}$. Hence, we can apply $exchange_{qr}$ and Lemma 2 in order to "exchange" q and r, and then apply $erase_q$ to erase q. Thus, we can construct a regular computation H' with induction number m, such that

- $\mathrm{Act}(H') = Y'$; (16)

Temp := H;
for each $p \in \mathrm{Act}(H) - Y'$ **do**
  Temp := $erase_p(Temp)$ **od**;
for each $q \in X_{j_{\max}}$ **do**
  if $q = \mathrm{Res}(H, v)$ for some v **then**
    Temp := $erase_q(Temp)$
  else
    — *By* (9), $\mathrm{Res}(H, v)$ *is nonempty; choose a process* $r \in \mathrm{Res}(H, v)$.
    Temp := $erase_q(exchange_{qr}(Temp))$
  fi od;
H' := *Temp*

Figure 7: Algorithm for constructing H' by erasing $\mathrm{Act}(H) - Y'$ and $X_{j_{\max}}$.

- $\mathrm{Cvr}(H') = \mathrm{Cvr}(H) - X_{j_{\max}}$;
- $V_C(H') = V_C(H)$;
- $\mathrm{Cvr}(H', v) \subseteq \mathrm{Cvr}(H, v)$ and $|\mathrm{Cvr}(H', v)| = c - 1$, for each $v \in V_C(H)$; (17)
- for each $p \in Y'$, v_p is not local to any process in $\mathrm{Cvr}(H')$;
- for each $p \in Y'$, either $v_p \in V_C(H')$ or no process in $\mathrm{Cvr}(H')$ writes to v_p in H'.

Also, by (9) and (17),

- $\mathrm{Res}(H', v)$ is nonempty, for each $v \in V_C(H)$. (18)

We now group processes in Y' depending on the variables accessed by their next remote events. For each $v \in V$, define Y_v as $\{p \in Y' \colon op(e_p) = \mathsf{write}(v) \text{ or } \mathsf{read}(v)\}$. Define V_{HC}, the set of variables that experience "high contention" (*i.e.*, those that are accessed by "sufficiently many" next remote events), as $V_{\mathrm{HC}} = \{v \in V \colon |Y_v| \geq 4 \log n\}$. Then, we have

$$|V_{\mathrm{HC}}| \leq \frac{|Y'|}{4 \log n}. \tag{19}$$

Define P_{HC}, the set of processes whose next remote event accesses a variable in V_{HC}, as $\bigcup_{v \in V_{\mathrm{HC}}} Y_v$. We now consider two cases, depending on $|P_{\mathrm{HC}}|$.

Case 1: $|P_{\mathrm{HC}}| < \frac{1}{2}|Y'|$ (erasing strategy)
Let $P_{\mathrm{LC}} = Y' - P_{\mathrm{HC}}$. Then, by (14), we have

$$|P_{\mathrm{LC}}| \geq |Y'|/2 \geq (n-1)/4c. \tag{20}$$

We construct an undirected graph $\mathcal{G} = (P_{\mathrm{LC}}, E_{\mathcal{G}})$. For each process p in P_{LC}, we introduce edge (p, q) (for some $q \in P_{\mathrm{LC}}$) if any of the following conditions holds: **(C1)** v_p is local to q; **(C2)** $v_p \notin V_C(H)$ and q writes to v in H'; **(C3)** $v_p \notin V_C(H)$ and e_q writes to v.

Because each variable is local to at most one process, Condition C1 can introduce at most one edge per process. By RC3, Condition C2 can introduce at most one edge per process.[8] Since $p \notin P_{\mathrm{HC}}$, we have $v_p \notin V_{\mathrm{HC}}$, and hence, Condition C3 can introduce at most $4 \log n - 1$ edge per process.

Thus, at most $4 \log n + 1$ edges are introduced per process, and hence the average degree of $\mathcal{G}$ is at most $8 \log n + 2$. Hence, by Theorem 4, there exists an independent set $\overline{Z} \subseteq P_{\mathrm{LC}}$ such that

$$|\overline{Z}| \geq \frac{|P_{\mathrm{LC}}|}{8 \log n + 3},$$

[8] In fact, By RC2, Condition C1 and C2 can *collectively* introduce at most one edge per process.

which, by (8) and (20), and using $8 \log n + 3 > 2 \log n + 1$, implies

$$|\overline{Z}| \geq \frac{n-1}{4(8 \log n + 3)^2}.$$

(Note that, since $n \geq 8 > 0$, we have $|\overline{Z}| > 0$, and hence $\overline{Z}$ is nonempty.)

Since a subset of an independent set is also an independent set, we can choose an independent set $Z \subseteq \overline{Z}$ such that

$$\frac{n-1}{4(8 \log n + 3)^2} \leq |Z| \leq n/4. \tag{21}$$

Next, we construct a computation G, satisfying the lemma, such that $\mathrm{Act}(G) = Z$.

Let $H'' = H' \mid (Z \cup \mathrm{Cvr}(H'))$. By (16), it follows that H'' is obtained from H' by erasing processes in $Y' - Z$, a subset of $\mathrm{Act}(H')$. Thus, by inductively applying $erase_p$ for each $p \in Y' - Z$, and using Lemma 1, it follows that H'' is a regular computation in C with induction number m, and

- $\mathrm{Res}(H'', v) = \mathrm{Res}(H', v)$, for each $v \in V_C(H)$. (22)

We now construct G^{m+1}, the $m+1^{\text{st}}$ segment of G, such that G is a regular computation in C, where $G = H'' \circ G^{m+1}$. Recall that G^{m+1} consists of four segments: $L^{m+1} \circ I^{m+1} \circ M^{m+1} \circ R^{m+1}$.

Define $V_C^{m+1}(G) = \{\}$, because no variables are newly covered. Thus, we have

- $V_C^{m+1}(G) = \{\}$, and $V_C^j(G) = V_C^j(H'') = V_C^j(H)$, for each $j = 1,\ 2,\ \ldots,\ m$. (23)

Let $d = |Z|$ and arbitrarily index the processes in Z as $Z = \{z_1,\ z_2,\ \ldots,\ z_d\}$. Define $L^{m+1} = L_{z_1} \circ L_{z_2} \circ \cdots \circ L_{z_d}$. Since H'' is regular and L^{m+1} is local, by RC2, it follows that for each variable v that is local to some process z in L^{m+1}, we have $value(v, H'') = value(v, (H \mid z))$. Thus, by (11), and repeatedly applying P2, we have $H'' \circ L^{m+1} \in C$. Since L^{m+1} is a local Z-computation, it is clearly a valid local segment.

We define M^{m+1} by concatenating the next remote events e_z, for $z \in Z$. By RC2, for each z and each local variable u of z, no process other than z itself writes to u in H''. Thus, we have $value(u, H'') = value(u, H \mid z)$.

By (15), either all next remote events of processes in Z are remote writes, or all are remote reads. In the former case, as shown above, each next event of Z cannot distinguish $H'' \circ L^{m+1}$ from $(H \mid z) \circ L_z$. Thus we have $H'' \circ L^{m+1} \circ M^{m+1} \in C$. In this case, we define $I^{m+1} = R^{m+1} = \langle\rangle$.

Otherwise, all next remote events of processes in Z are remote reads. By (18) and (22), for each variable $v \in V_C(H)$, $\mathrm{Res}(H'', v)$ is nonempty. Hence, we can choose a process q_v in $\mathrm{Res}(H'', v)$ to cover v in G^{m+1}. By RC5, we have $(H'' \mid q_v) \circ \langle e_{q_v}, f_{q_v} \rangle \in C$, where e_{q_v} (resp. f_{q_v}) is an invocation (resp. response) event on v. We define I^{m+1} (resp. R^{m+1}) by concatenating the events e_{q_v} (resp. f_{q_v}) where v ranges over $V_C(H)$. Since each variable is properly covered, it can be shown that $G = H'' \circ L^{m+1} \circ I^{m+1} \circ M^{m+1} \circ R^{m+1} \in C$.

In either case, conditions RC1–RC5 can be individually checked to hold in G (with '$\mathrm{Act}(G)' \leftarrow Z$), which implies that G is a regular computation with induction number $m+1$. By (21), we have (4). By (21), we have $\log \mathrm{Act}(G) \leq \log n - 2$. Also, by (21), we have $\log|Z| \leq \log n - 2$, and combining with (17), we have (6). It follows that G is a computation that satisfies the lemma.

Case 2: $|P_{\mathrm{HC}}| \geq \frac{1}{2}|Y'|$ (covering strategy)

For each $v \in V_{\mathrm{HC}}$, define x_v, the process in P_{HC} that conflicts with Y_v, as follows:

- **(K1)** If v is local to a process $q \in P_{\mathrm{HC}}$, then let $x_v = q$.
- **(K2)** Otherwise, if $v \in V_C(H')$, then let $x_v = \perp$.
- **(K3)** Otherwise, if there exists a write to v in H', then only one process q writes to v in H'. (Note that, since H' is regular, by RC3, if multiple processes write to v in H', then $v \in V_C(H')$, *i.e.*, (K2) holds.) If $q \in P_{\mathrm{HC}}$, then let $x_v = q$; otherwise, let $x_v = \perp$.
- **(K4)** If none of the above holds, then let $x_v = \perp$.

Define K, the erased (or "killed") processes, as $K = \{x_v : v \in V_{\mathrm{HC}} \text{ and } x_v \neq \perp\}$. Since $|K| \leq |V_{\mathrm{HC}}|$, by (19), we have

$$|K| \leq |Y'|/(4 \log n). \tag{24}$$

Recall that $P_{\mathrm{HC}} = \bigcup_{v \in V_{\mathrm{HC}}} Y_v$, *i.e.*, P_{HC} consists of disjoint subsets Y_v for each $v \in V_{\mathrm{HC}}$, and each such Y_v contains at least $4 \log n$ processes (by the definition of V_{HC}).

The processes in K are precisely those that conflicts with other processes in P_{HC} and hence must be eliminated. However, after erasing K, some set Y_{v_k} (for some $v_k \in V_{\mathrm{HC}}$) may have only $o(\log n)$ processes left, in which case we do not have enough processes to cover v_k. We solve this problem by simply erasing any such Y_{v_k}, as follows.

Define V_K, the variables that do not have enough covering processes (and hence must be "killed"), as $V_K = \{v \in V_{\mathrm{HC}} : |Y_v - K| \leq 3 \log n\}$. Define K', the secondary set of killed processes, as $K' = \bigcup_{v \in V_K} Y_v$. Define S, the "survivors," as $S = P_{\mathrm{HC}} - K - K'$. By the definition of P_{HC} and K',

$$S = (P_{\mathrm{HC}} - K') - K = \big(\bigcup_{v \in V_{\mathrm{HC}} - V_K} Y_v\big) - K. \tag{25}$$

Consider each $v_k \in V_K$. Define $n(v_k) = |Y_{v_k}|$. By the definition of V_K, we have $n(v_k) - |Y_{v_k} \cap K| = |Y_{v_k} - K| \leq 3 \log n$. By the definition of V_{HC}, we also have $n(v_k) \geq 4 \log n$. By dividing the former inequality by the latter, we have $1 - |Y_{v_k} \cap K|/n(v_k) \leq 3/4$, and hence,

$$n(v_k) \leq 4 \cdot |Y_{v_k} \cap K|. \tag{26}$$

Note that, since $K \subseteq P_{\mathrm{HC}}$, we have $K = \bigcup_{v \in V_{\mathrm{HC}}} (Y_v \cap K)$, and hence $|K| = \sum_{v \in V_{\mathrm{HC}}} |Y_v \cap K|$. Therefore, by the definition of K',

$$\begin{aligned} |K| &\geq \textstyle\sum_{v \in V_K} |Y_v \cap K|, \text{ and} \\ |K'| &= \textstyle\sum_{v \in V_K} n(v_k). \end{aligned}$$

Combining these equations with (26), we have

$$|K'| \leq 4|K|. \tag{27}$$

By (8), (14), (24), (27), and $|P_{\mathrm{HC}}| \geq \frac{1}{2}|Y'|$ (which is assumed in Case 2), we have

$$\begin{aligned} |S| &\geq |P_{\mathrm{HC}}| - |K'| - |K| \\ &\geq |P_{\mathrm{HC}}| - 5|K| \\ &\geq |Y'| \cdot (1/2 - 5/(4 \log n)) \\ &\geq \frac{(n-1)(2 \log n - 5)}{8c \log n} \\ &\geq \frac{(n-1)(2 \log n - 5)}{8 \log n (2 \log n + 1)}, \end{aligned}$$

and hence, since $\log n < 2\log n + 1$,

$$|S| > \frac{(n-1)(2\log n - 5)}{8(2\log n + 1)^2}. \quad (28)$$

Define $H'' = H \mid (S \cup \mathrm{Cvr}(H'))$. By (16) and $P_{\mathrm{HC}} \subseteq Y'$, it follows that H'' is obtained from H' by erasing processes in $Y' - S$, a subset of $\mathrm{Act}(H')$. Thus, by inductively applying $erase_p$ for each such p, and using Lemma 1, it follows that H'' is a regular computation in C with induction number m, and $\mathrm{Res}(H'', v) = \mathrm{Res}(H', v)$, for each $v \in V_C(H)$.

Define V_S, the surviving variables, as $V_S = \{v \in V_{\mathrm{HC}}\colon S \cap Y_v \neq \{\}\}$. By the definitions of P_{HC} and K', we have

- $|Y_v \cap S| > 3\log n$ for each variable $v \in V_S$. (29)

We now construct G^{m+1}, the $m+1^{\mathrm{th}}$ segment of G, such that G is a regular computation in C, where $G = H'' \circ G^{m+1}$. Recall that G^{m+1} consists of four segments: $L^{m+1} \circ I^{m+1} \circ M^{m+1} \circ R^{m+1}$.

If the next remote events of S consist solely of remote reads, then we can construct G^{m+1} in the same way as in Case 1. Otherwise, the remote events of S consist solely of remote writes. Let $V_C^{m+1}(G) = V_S$, *i.e.*, all variables that are remotely accessed by next remote events are now covered. For each such variable $v \in V_S$, choose $\lceil 2\log n \rceil$ processes in Y_v, and define $\mathrm{Cvr}(G, v)$ to be those $\lceil 2\log n \rceil$ processes. By (29), it follows that we can choose $\mathrm{Cvr}(G, v)$ for each v and still have at least $|S|/3$ processes not selected as covering processes; these processes constitute $\mathrm{Act}(G)$. Thus, by (28), we have

$$|\mathrm{Act}(G)| \geq \frac{(n-1)(2\log n - 5)}{24\log n(2\log n + 1)},$$

which satisfies (4). The construction of G^{m+1} is similar to Case 1, except that we use 'Z' $\leftarrow S$ and '$\mathrm{Cvr}(H)$' $\leftarrow \mathrm{Cvr}(H) \cup \big(\bigcup_{v \in V_S} \mathrm{Cvr}(G, v)\big)$. □

THEOREM 2. *For any mutual exclusion system $\mathcal{S} = (C, P, V)$, there exist a p-computation F such that F does not contain CS_p, and p executes $\Omega(\log N/\log\log N)$ remote events in F, where $N = |P|$.*

Proof: Let $H_1 = \langle Enter_1, Enter_2, \ldots, Enter_N \rangle$, where $P = \{1, 2, \ldots, N\}$. By the definition of a mutual exclusion system, $H_1 \in C$. It is obvious that H_1 is a regular computation with induction number 1, such that $\mathrm{Act}(H_1) = P$, $\mathrm{Cvr}(H_1) = \{\}$, and $V_C(H_1) = \{\}$. We repeatedly apply Lemma 4 and construct a sequence of computations H_1, H_2, $\ldots$, H_k, such that each computation H_j has induction number j. Define $n_j = |\mathrm{Act}(H_j)|$. Then, by (4),

$$n_{j+1} \geq \min\left(\frac{(n_j - 1)(2\log n_j - 5)}{24(2\log n_j + 1)^2}, \frac{n_j - 1}{4(8\log n_j + 3)^2}\right),$$

which implies

$$n_{j+1} \geq \frac{a n_j}{\log^2 n_j} \geq \frac{a n_j}{\log^2 N},$$

where a is some fixed constant. This in turn implies

$$\log n_{j+1} \geq \log n_j - 2\log\log N + \log a.$$

Therefore, by iterating over $1 \leq j < k$, and using $n_1 = N$, we have

$$\log n_k \geq \log N - 2(k-1)\log\log N + (k-1)\log a. \quad (30)$$

We stop the induction at step k when Assumption (1) in Lemma 4 is not satisfied, in which case we have established $k > \log|\mathrm{Act}(H_k)| - 2$. Combining this inequality with (30), we have

$$k > \frac{\log N + 2\log\log N - \log a - 2}{2\log\log N - \log a + 1} = \Theta\left(\frac{\log N}{\log\log N}\right).$$

By RC5, $H_k \mid p$ is a valid computation in C. Since each process p in $\mathrm{Act}(H_k)$ executes exactly k remote events in H_k, $H_k \mid p$ is a computation that satisfies the theorem. □

We now prove Theorem 3. We henceforth consider a system in which Properties P2 and P4 are replaced by P2′ and P4′, respectively (see Sec. 4).

We will only explain the part of the proof that is different from the proof of Theorem 2. Since we now allow concurrent writes of the same variable, for the sake of simplicity, we assume that no covering writes terminate, *i.e.*, any invocation event in a regular computation H is the last event by that process in H. (This assumption is not really needed, but is being made here to make the reasoning a bit easier.) Therefore, we change the definition of a regular computation such that response segments are now defined to be empty. We also replace both RC1 and RC3 by the following single condition RC1′:

RC1′: Assume that H can be written as $\langle \ldots, e_p, \ldots, f_q, \ldots \rangle$. If $p \neq q$ and there exists a variable $v \in Wvar(e_p) \cap var(f_q)$ such that e_p is the first write to v in H, then $v \in V_C(H)$ and e_p is a (non-terminating) invocation event on v. — *Informally, if p is the first process to write variable v, and if another process q later accesses v, then p covers v throughout the rest of H, and hence any read from v after e_p can read any value.*

The next lemma is a variation of Lemma 3 that deals with critical remote events. It is adapted from Lemma 8 of [8].

Lemma 5 *Let H be a regular computation in C with induction number m. Define $n = |\mathrm{Act}(H)|$. Then, there exists another regular computation H' in C with induction number m and a subset Y of $\mathrm{Act}(H)$, satisfying the following:*

- *H' contains all events in H;*
- *$n - 1 \leq |Y| \leq n$;*
- *for each process p in Y, there exist a critical event e_p of p, such that $H' \circ \langle e_p \rangle \in C$ and $e_p \neq CS_p$.*

Proof: We can show that Lemma 3 still holds with the modified definitions. Therefore, Lemma 3 implies that there exists a subset Y of $\mathrm{Act}(H)$ such that $n - 1 \leq |Y| \leq n$ and each process in Y has a remote event after H, *i.e.*, for each $p \in Y$, there exist a local p-computation L_p and a remote write or remote read event e_p, such that $(H \mid p) \circ L_p \circ \langle e_p \rangle \in C$ and $e_p \neq CS_p$.

If all remote events e_p (for $p \in Y$) are critical after H, then define $M^m_{(H')}$, the m^{th} remote segment of H', as $M^m_{(H')} = M^m \circ L_{p_1} \circ L_{p_2} \circ \cdots \circ L_{p_d}$, where M^m is the m^{th} remote segment of H and $Y = \{p_1, p_2, \ldots, p_d\}$. (Note that a remote segment is allowed to contain *local* events.) Define all other segments of H' to be the same as in H, *i.e.*, $H' = H \circ L_{p_1} \circ \cdots \circ L_{p_d}$. It can be shown that H' is a regular computation that satisfies Lemma 5.

Otherwise, assume that some event e_p is noncritical after H, and that e_p accesses a remote variable v. Then, we have

$$H \circ L_p \circ \langle e_p \rangle = E \circ \langle f_p \rangle \circ F \circ L_p \circ \langle e_p \rangle, \tag{31}$$

where f_p is the last event by p that accesses v, and

- $F \mid p$ contains no expanding reads or writes. (32)

Let $G = E \circ \langle f_p \rangle \circ (F \mid p) \circ L_p \circ \langle e_p \rangle \circ (F - F \mid p)$. In the following paragraphs, we show that G is a regular computation in C. Observe that G contains all events contained in H and more remote events than H. By the Progress requirement, this implies that we can apply this argument only a finite number of times, *i.e.*, if we repeatedly apply Lemma 3 and construct a new computation in the manner in which G is constructed, then we eventually obtain a computation H' such that applying Lemma 3 yields at least $n-1$ processes in $\mathrm{Act}(H)$, each of which has a *critical* remote event after H'. By our construction, H' is a regular computation in C that contains all events contained in H.

To begin the construction of G, note that, because $H \in C$, (31) implies $H = E \circ \langle f_p \rangle \circ F \in C$. We claim that $E \circ \langle f_p \rangle \circ (F \mid p) \in C$ holds. Note that, by RC1′, if an event g_p in $F \mid p$ reads a remote variable u, then one of the following holds: **(i)** u is not written by any other process before g_p, **(ii)** $u \in V_C(H)$ and E does not contain a write to u, or **(iii)** $u \in V_C(H)$ and there exists a write to u in E.

If either (i) or (ii) holds, then it is clear that g_p cannot distinguish between F and $F \mid p$. On the other hand, if (iii) holds, then by RC1′, the first write to u (in E) covers u, so g_p cannot distinguish between F and $F \mid p$. Thus, no events in $F \mid p$ may distinguish between F and $F \mid p$ after $E \circ \langle f_p \rangle$. Thus we have $E \circ \langle f_p \rangle \circ (F \mid p) \in C$.

Since L_p is a local computation, by applying P2, we can easily show $E \circ \langle f_p \rangle \circ (F \mid p) \circ L_p \in C$. (Note that, by RC2, no process other than p may write p's local variables in E.) Similarly, since e_p is not an expanding event, by applying P2, we can show $E \circ \langle f_p \rangle \circ (F \mid p) \circ L_p \circ \langle e_p \rangle \in C$. (If e_p writes to v, then e_p reads only local variables. On the other hand, if e_p reads v, then since e_p is not an expanding read, by (32), E contains a read from v by p. Thus, either no other process writes to v in E, or v is covered at the end of E.)

We now claim $G \in C$. It suffices to show the following:

- **Claim 1:** If g_p is an event of $F \mid p$, h_q is an event of $F - F \mid p$, and u is a variable in $Wvar(g_p) \cap Rvar(h_q)$, then some event in E covers u. (In other words, no event in $F - F \mid p$ can distinguish between H and G.)

  Proof of Claim: Consider each variable u in $Wvar(g_p)$ or $Rvar(h_q)$. By RC2, u cannot be local to either p or q. On the other hand, if u is remote to both p and q, then since g_p is not an expanding write, by (32), E contains an event $\bar{g}_p$ by p that writes to u. Since h_q is an event of F, H can be written as $\langle \ldots, \bar{g}_p, \ldots, h_q, \ldots \rangle$. Therefore, by applying RC1′, it follows that the first event to write to u in H (which must precede $\bar{g}_p$) covers u. □

Thus, it follows that G is a computation in C that contains all events in H. The conditions RC1′, RC2, RC4, and RC5 can be individually checked to hold in G. Therefore, by inductively applying the construction of G, we can construct H', as explained above. □

Finally, a variation of Lemma 4 can be proved using Lemma 5 instead of Lemma 3. Apart from the fact that P2′, P4′, and RC1′ are now being used, the only changes to Lemma 4 that are needed are the following.

- Assertion (2) in the statement of the lemma changes to: each process in $\mathrm{Act}(H)$ executes *at least* one remote event in each remote segment M^j, for $j = 1, \ldots, m$.
- Assertion (5) in the statement of the lemma changes to: each process in $\mathrm{Act}(G)$ executes *at least* one remote event in each remote segment M^j, for $j = 1, \ldots, m+1$;

Given this new lemma, Theorem 3 can be proved in a way that is identical to Theorem 2. Thus, we have the following.

THEOREM 3. *For any mutual exclusion system $\mathcal{S} = (C, P, V)$ that satisfies Properties P2′ and P4′ instead of P2 and P4, there exist a p-computation F such that F does not contain CS_p, and p accesses $\Omega(\sqrt{\log N / \log\log N})$ distinct remote variables in F, where $N = |P|$.* □