Time Bounds for Mutual Exclusion and Related Problems*
(Extended Abstract)

Jae-Heon Yang
Department of Computer Science
The University of Maryland
College Park, Maryland 20742-3255

James H. Anderson
Department of Computer Science
The University of North Carolina
Chapel Hill, North Carolina 27599-3175

Abstract

We establish trade-offs between time complexity and write- and access-contention for solutions to the mutual exclusion problem. The *write-contention* (*access-contention*) of a concurrent program is the number of processes that may be simultaneously enabled to write (access) the same shared variable. Our notion of time complexity distinguishes between local and remote references to shared memory. We show that, for any N-process mutual exclusion algorithm with write-contention w, there exists an execution involving only one process in which that process executes $\Omega(\log_w N)$ remote memory references for entry into its critical section. We further show that among these remote references, $\Omega(\sqrt{\log_w N})$ distinct remote variables are accessed. For algorithms with access-contention c, we show that the latter bound can be improved to $\Omega(\log_c N)$. The last two of these results imply that a trade-off between contention and time complexity exists even if coherent caching techniques are employed. Because the execution that establishes these bounds involves only one process, our results show that "fast mutual exclusion" requires arbitrarily high write-contention. We show that these bounds hold when using any of a variety of synchronization primitives, including read, write, test-and-set, load-and-store, compare-and-swap, and fetch-and-add, and that they can be generalized to apply when using even stronger primitives. Our results can be extended to apply to a class of decision problems that includes the leader-election problem. The time bounds that we establish are the first of their kind for asynchronous shared memory concurrent programs.

*Work supported, in part, by NSF Contracts CCR-9109497 and CCR-9216421 and by the NASA Center for Excellence in Space Data and Information Sciences (CESDIS).

1 Introduction

The mutual exclusion problem is a fundamental paradigm for coordinating accesses to shared data on asynchronous shared memory multiprocessing systems [4]. In this problem, accesses to shared data are abstracted as "critical sections" of code, and it is required that at most one process executes its critical section at any time. In this paper, we consider bounds on time for mutual exclusion, a subject that has received scant attention in the literature. Past work on the complexity of mutual exclusion has almost exclusively focused on space requirements; the limited work on time bounds that has been done has focused on partially synchronous models [10].

The lack of prior work on time bounds for mutual exclusion within asynchronous models is probably due to difficulties associated with measuring the time spent within busy-waiting constructs. In fact, because of such difficulties, there has been scarcely little work of any kind on time bounds for asynchronous concurrent programming problems for which busy-waiting is inherent. One of the primary contributions of this paper is to show that it *is* possible to establish meaningful time bounds for such problems.

A natural approach to measuring the time complexity of a mutual exclusion algorithm would be to simply use the standard sequential programming measure of counting all operations. However, in any algorithm in which processes busy-wait, the number of operations needed for one process to get to its critical section is unbounded in the worst case. In other words, the standard sequential programming metric yields no useful information concerning the performance of such algorithms under contention.

In a recent paper, Yang and Anderson proposed a time measure for concurrent programs that distinguishes between local and remote accesses of shared memory [14]. A shared variable access is *local* if does not

require a traversal of the global interconnect between processors and shared memory, and is *remote* otherwise. Although the notion of a *locally* accessible *shared* variable may seem counterintuitive, there are two architectural paradigms that support it. In particular, on distributed shared memory machines, a shared variable can be made locally accessible by storing it in a local portion of shared memory, and on cache-coherent machines, a shared variable can become locally accessible by migrating to a local cache line.

Under Yang and Anderson's proposed measure, the time complexity of a concurrent program is measured by counting only remote accesses of shared variables; local accesses are ignored. This measure satisfies two criteria that must be met by any reasonable complexity measure. First, it is conceptually simple. In fact, this measure is a natural descendent of the standard time complexity measure used in sequential programming. Second, this measure has a tangible connection with real performance, as demonstrated by a number of recently published performance studies [3, 11, 14]. All other proposed time complexity measures for concurrent programs that we know of fail to satisfy at least one of these criteria.[1]

We present several lower-bound results for mutual exclusion that are based on the time complexity measure of Yang and Anderson. Our results establish tradeoffs between time complexity and write- and access-contention for solutions to the mutual exclusion problem. The *write-contention* (*access-contention*) of a concurrent program is the number of processes that may be simultaneously enabled to write (access) the same shared variable. Limiting access-contention is an important consideration when designing algorithms for problems, such as mutual exclusion and shared counting, that must cope well with high competition among processes [3, 7, 8, 13]. Performance problems associated with high access-contention can be partially alleviated by employing coherent caching techniques to reduce concurrent reads of the same memory location. However, even when such techniques are employed, limiting write-contention is still an important concern.

We show that, for any N-process mutual exclusion algorithm with write-contention w, there exists an execution involving only one process in which that process executes $\Omega(\log_w N)$ remote memory references for entry into its critical section. We further show that among these remote references, $\Omega(\sqrt{\log_w N})$ distinct remote variables are accessed. For algorithms with access-contention c, we show that the latter bound can be improved to $\Omega(\log_c N)$.

[1] Our time complexity measure cannot be used to make distinctions between programs that busy-wait on remote variables. However, many concurrent programming problems that require busy-waiting (including mutual exclusion) can be solved without busy-waiting on such variables.

These results have a number of important implications. For example, because the first access of any variable causes a cache miss, the latter two bounds imply that a time/contention trade-off exists even if coherent caching techniques are employed. Also, because the execution that establishes these bounds involves only one process, it follows that so-called fast mutual exclusion algorithms — i.e., algorithms that require a process to execute only a constant number of remote memory references in the absence of competition [9] — require arbitrarily high write-contention in the worst case. These bounds hold assuming that each atomic operation accesses at most one remote variable. A variety of well-known synchronization primitives satisfy this assumption, including read, write, test-and-set, load-and-store, compare-and-swap, and fetch-and-add. We show that our basic results can be extended to apply to programs with operations that access multiple remote variables, establishing similar trade-offs between time complexity and atomicity. Our results apply not only to mutual exclusion but also to a class of decision problems that includes the leader-election problem.

Related work includes previous research by Dwork et al. given in [5], where it is shown that solving mutual exclusion with access-contention c requires $\Omega((\log_2 N)/c)$ memory references. Our work extends that of Dwork et al. in several directions. First, the implications concerning fast mutual exclusion and cache coherence noted above do not follow from their work. Second, we consider programs in which atomic operations may access multiple shared variables, whereas they only consider reads, writes, and read-modify-writes. Third, in our main result, we restrict only write-contention and obtain a tight bound of $\Omega(\log_w N)$, which exceeds the bound established by them. Finally, and most importantly, Dwork et al. make no distinction between local and remote shared memory accesses. Because busy-waiting is required for mutual exclusion in general, an unbounded number of memory accesses (local or remote) are required in the worst case. It is our belief that time complexity results that do not distinguish between local and remote accesses to shared memory are of questionable value as a measure of performance of mutual exclusion algorithms under contention.

The rest of the paper is organized as follows. In Section 2, we present our model of shared memory systems. The above-mentioned time bounds are then established in Section 3. Concluding remarks appear in Section 4.

2 Shared Memory Systems

Our model of a shared memory system is similar to that given by Merritt and Taubenfeld in [12]. A *system* $S = (C, P, V)$ consists of a set of computations C, a set of processes $P = \{1, 2, \cdots, N\}$, and a set of variables V.

A *computation* is a finite sequence of events.

An *event*, denoted $[R, W, i]$, where $R = \{(x_j, u_j) | 1 \leq j \leq m\}$ for some m, $W = \{(y_k, v_k) | 1 \leq k \leq n\}$ for some n, and $i \in P$, represents reading value u_j from variable x_j, for $1 \leq j \leq m$, and writing value v_k to variable y_k, for $1 \leq k \leq n$; each variable in R (W) is assumed to be distinct. We say that this event *accesses* each such x_j and y_k. We use $R.var$ to denote the set of variables x_j such that $(x_j, u_j) \in R$ for some u_j, and $W.var$ to denote the set of variables y_k such that $(y_k, v_k) \in W$ for some v_k. An *initial value* is associated with each variable.

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

We use $\langle e, \cdots \rangle$ to denote a computation that begins with the event e, and $\langle\rangle$ to denote the empty computation. We define the *length* of computation H, denoted $|H|$, as the number of events in H. $H \circ G$ denotes the computation obtained by concatenating computations H and G. If G is a subsequence of H, then $H - G$ is the computation obtained by removing all events in G from H. The value of variable x at the end of computation H, denoted $value(x, H)$, is the last value that is written to x in H (or the initial value of x if x is not written in H).

An *extension* of computation H is a computation of which H is a prefix. For a computation H and a set of processes Y, H_Y denotes the subsequence of H that contains all events in H of processes in Y. A computation H is a *Y-computation* iff $H = H_Y$ holds.

Computations H and G are *equivalent* with respect to a set of processes Y, denoted $H[Y]G$, iff $H_Y = G_Y$. Note that $[Y]$ is an equivalence relation. For simplicity, we abbreviate the preceding definitions when applied to a singleton set of processes. For example, if $Y = \{i\}$, then we use H_i to mean $H_{\{i\}}$, i-computation to mean $\{i\}$-computation, and $[i]$ to mean $[\{i\}]$. We now present our model of shared memory systems.

Definition: A *shared memory system* $S = (C, P, V)$ is a system that satisfies the following properties.

- (P1) If $H \in C$ and G is a prefix of H, then $G \in C$.
- (P2) If $H \circ \langle [R, W, i] \rangle \in C$, $G \in C$, $G[Y]H$, and $i \in Y$, and if for all $x \in R.var$, $value(x, H) = value(x, G)$ holds, then $G \circ \langle [R, W, i] \rangle \in C$.
- (P3) If $H \circ \langle [R, W, i] \rangle \in C$, $G \in C$, $G[Y]H$, and $i \in Y$, then $G \circ \langle [R', W', i] \rangle \in C$ for some R' and W' such that $R'.var = R.var$ and $W'.var = W.var$.
- (P4) For any $H \in C$, $H \circ \langle [R, W, i] \rangle \in C$ only if for all $(x, v) \in R$, $v = value(x, H)$ holds. □

Most of our results are dependent on the following assumption concerning events. Note that this assumption defines the allowable degree of "atomicity" for events.

Atomicity Assumption: Each event of process i may access at most one variable that is remote to i. □

For simplicity, we call a remote event a *remote read* if it does not write a remote variable, and a *remote write* otherwise. Note that a remote write may both read and write the remote variable that it accesses. A wide variety of synchronization primitives satisfy the Atomicity Assumption, including read, write, test-and-set, load-and-store, compare-and-swap, and fetch-and-add. In Section 3.3, we show that our results can be generalized by relaxing this assumption.

In the following section, we establish time bounds involving various notions of contention. Consider a shared memory system $S = (C, P, V)$. The strictest notion of contention is static in nature. In particular, consider a variable x in V. A process i in P is a *reader* (*writer*) of x iff there is an event of i that reads (writes) x in some computation in C. We say that x is a *k-reader* (*k-writer*) *variable* iff there are k readers (writers) of x. The other two notions of contention that we employ are dynamic in nature. For $H \in C$ and $x \in V$, let $overwriters(x, H) \equiv \{i \mid H \circ \langle [R, W, i] \rangle \in C$ where $x \in W.var\}$. Then, the *write-contention* of S is $\mathbf{max}_{x \in V, H \in C}(|overwriters(x, H)|)$. Similarly, let $contenders(x, H) \equiv \{i \mid H \circ \langle [R, W, i] \rangle \in C$ where $x \in (R.var \cup W.var)\}$. Then, the *access-contention* of S is $\mathbf{max}_{x \in V, H \in C}(|contenders(x, H)|)$. These notions of contention bound the number of processes that may simultaneously write (access) the same memory location.

3 Time Bounds for Mutual Exclusion

Our main results concerning the mutual exclusion problem are based on a simplified version of the problem, which we call the "minimal mutual exclusion problem".

Minimal Mutual Exclusion Problem: We define the minimal mutual exclusion problem for a shared memory system $S = (C, P, V)$ as follows. Each process $i \in P$ has a local variable $i.dine$ that ranges over $\{think, hungry, eat\}$. Variable $i.dine$ is initially $think$ and is accessed only by the following events:

$$\begin{aligned} Think_i &\equiv [\{\}, \{(i.dine, think)\}, i] \\ Hungry_i &\equiv [\{\}, \{(i.dine, hungry)\}, i] \\ Eat_i &\equiv [\{\}, \{(i.dine, eat)\}, i] \end{aligned}$$

The allowable transitions of $i.dine$ are as follows: for any $H \in C$, $H \circ \langle Think_i \rangle \in C$ iff $value(i.dine, H) = eat$;

$H \circ \langle Hungry_i \rangle \in C$ iff $value(i.dine, H) = think$; and if $H \circ \langle Eat_i \rangle \in C$, then $value(i.dine, H) = hungry$. System S solves the minimal mutual exclusion problem iff the following requirements are satisfied.

- *Exclusion*: For any $H \in C$ and processes $i \neq j$, $value(i.dine, H) = eat \Rightarrow value(j.dine, H) \neq eat$.
- *Progress*: For any $H \in C$ and process $i \in P$, if H is an i-computation, then either H contains Eat_i, or there exists an i-computation G such that $H \circ G \circ \langle Eat_i \rangle \in C$. □

Note that the Progress condition above is much weaker than that usually specified for the mutual exclusion problem. (This, of course, strengthens our impossibility proofs.)

Before presenting our main results, we give bounds for the case of statically-defined contention. In this theorem and those that follow, we assume that S is a shared memory system and that $i \in P$.

Theorem 1: If $S = (C, P, V)$ solves the minimal mutual exclusion problem, and if either all variables in V are k-reader variables, or all variables in V are k-writer variables, then there exists an i-computation in C that contains $\Omega(N/k)$ remote events but no Eat_i event.

Proof Sketch: It can be shown that for any i-computation H containing Eat_i and any process $j \neq i$, H contains a read of a variable that can be written by j and a write of a variable that can be read by j. By employing this fact, it is possible to prove the theorem by using a relatively simple counting argument. □

For any N-process system S that satisfies the conditions of Theorem 1, some process i executes $\Omega(N/k)$ remote events in the absence of competition. If we remove process i from system S, we obtain a system that satisfies the conditions of the theorem with N replaced by $N-1$. Thus, there is a process $j \neq i$ in system S that executes $\Omega((N-1)/k)$ remote events in the absence of competition. Continuing in this manner, at least half the processes in S execute at least $\Omega(N/2k)$ remote events in the absence of competition. Thus, we have the following corollary.

Corollary 1: For any system S satisfying the conditions of Theorem 1, there exist $\Omega(N)$ processes i in P for which the conclusion of the theorem holds. □

Similar corollaries apply to the theorems in the following subsections.

In [2], a mutual exclusion algorithm requiring $O(N)$ remote memory references per critical section acquisition is given that employs only single-reader, single-writer variables. Thus, if k is taken to be a positive constant, then the bound of Theorem 1 is asymptotically tight. In the remainder of the paper, we consider more interesting bounds based on dynamic notions of contention.

3.1 Main Result: Bounding Remote Events

In this section, we show that for any system with write-contention w, $\Omega(log_w N)$ remote events are required in the absence of competition to solve the minimal mutual exclusion problem. This bound has important consequences for distributed shared memory multiprocessing systems. On such systems, remote events require a traversal of a global interconnection network and hence are more expensive than local events. Thus, for such machines, the lower bound of Theorem 2 not only gives the inherent time complexity of the problem, it also bounds the communication complexity measured in terms of global traffic.

Theorem 2 is proved by considering a class of computations, as defined by a set of conditions. Each of these conditions refers to an arbitrary computation H in this class. The first condition is as follows.

- (C1) For events $[R, U, i]$ and $[T, W, j]$ in H, if $(R.var \cap W.var) \neq \{\}$ holds and $[T, W, j]$ precedes[2] $[R, U, i]$ in H, then $i = j$. Informally, no process reads a variable that is accessed by a preceding write of another process in H.

The next lemma gives us a means for projecting a computation onto a set of processes so that the resulting projection is itself a computation.

Lemma 1: For any $S = (C, P, V)$, if a computation H in C satisfies (C1), then for any $Y \subseteq P$, $H_Y \in C$.

Proof Sketch: For any process in Y, H_Y is not distinguishable from H. Thus, we can let processes in Y execute the same events as they execute in H. □

Before presenting the remaining lemmas, we state the remaining three conditions that serve to characterize the class of computations considered in the main theorem. The first of these conditions refers to "active" processes. If $H = \langle\rangle$ or $H_i \neq \langle\rangle$, then process i is *active* in H; otherwise i is *inactive* in H. Recall that in these conditions, H denotes an arbitrary computation from the class to be considered.

- (C2) For any event $[R, W, i]$ in H, if $x \in (R.var \cup W.var)$, and if x is local to a process j that is active

[2] Although our definition of an event allows multiple instances of the same event, we assume that such instances are distinguishable from each other. (For simplicity, we do not extend our notion of an event to include an additional identifier for distinguishability.)

in H, then $i = j$. Informally, no local variable of an active process is accessed by other processes in H.

- (C3) For any events $[R, W, i]$ and $[T, U, j]$ in H, if $(W.var \cap U.var) \neq \{\}$, then $i = j$. Informally, each variable is written by at most one process in H.
- (C4) For any prefix G of H, $value(i.dine, G) \neq eat$. Informally, no process eats in H.

According to the next lemma, if n processes are competing for entry into their critical sections, and if each of these n processes has no knowledge of the others, then at least $n-1$ of the processes have at least one more remote event to execute. To formally capture the latter, consider a system $S = (C, P, V)$ that solves the minimal mutual exclusion problem and let $i \in P$ and $H \in C$. We say that *i has a remote event after H* iff there exists an i-computation M such that M does not contain Eat_i, M has a remote event, and $H \circ M \in C$.

Lemma 2: Suppose that $S = (C, P, V)$ solves the minimal mutual exclusion problem. Let $Y \subseteq P$ be a set of n processes, and let H be a Y-computation in C satisfying (C1), (C2), and (C4). Then, at least $n-1$ processes in Y have a remote event after H.

Proof Sketch: If there are two processes that do not have a remote event after H, then we can extend H by executing those processes and violate the Exclusion requirement. □

Our next lemma provides the induction step that leads to the lower bound in Theorem 2.

Lemma 3: Let $S = (C, P, V)$ be a shared memory system with write-contention w that solves the minimal mutual exclusion problem. Let $Y \subseteq P$ be a set of n processes, and let H be a Y-computation in C satisfying (C1), (C2), (C3), and (C4) such that each process in Y executes r remote events in H. Then, there exist $Z \subseteq Y$, where $|Z| = \lceil (n-1)/6w \rceil$, and a Z-computation G in C satisfying (C1), (C2), (C3), and (C4) such that each process in Z executes $r+1$ remote events in G.

Proof Sketch: The proof strategy is as follows. We show that there exists $Z \subseteq Y$ such that each process in Z can execute another remote event without violating any of the conditions (C1) through (C4). We "eliminate" processes not in Z, i.e., ones that may violate some condition. Finally, we construct a Z-computation G that satisfies (C1), (C2), (C3), and (C4).

Lemma 2 implies that there exists $Y1 \subseteq Y$, where $|Y1| \geq n-1$, satisfying the following condition: for any $i \in Y1$, there exists an i-computation $B(i)$ such that $H \circ B(i) \in C$, $B(i)$ does not contain Eat_i, and $B(i)$ has at least one remote event. For $i \in Y1$, let $B(i) = L(i) \circ \langle [R_i, W_i, i], \cdots \rangle$ where $[R_i, W_i, i]$ is the first remote event in $B(i)$.

By the Atomicity Assumption, each remote event accesses exactly one remote variable. Let X denote the set of variables that are accessed as remote variables by the events $[R_i, W_i, i]$ where $i \in Y1$. For any $x \in X$, let $Q_x = \{i \mid x \in W_i.var \text{ and } x \text{ is remote to } i\}$. Informally, Q_x consists of all those processes that may be simultaneously enabled to write x after H. Using the fact that S has write-contention w, it is possible to show that $|Q_x| \leq w$.

From each Q_x, where $x \in X$, we eliminate all but one process. We use $Y2$ to denote the set of processes that are not eliminated from $Y1$. Note that, from the Atomicity Assumption, for each distinct x and y in X, $Q_x \cap Q_y = \{\}$. Thus, because $|Q_x| \leq w$, it follows that $|Y2| \geq \lceil (n-1)/w \rceil$.

In order to identify those processes that must be eliminated from $Y2$, we construct an undirected graph $\langle Y2, E \rangle$ as follows. We do not distinguish a vertex representing p from the process p when this does not cause any confusion. Informally, an edge joining two processes represents possible information flow between the two processes. Our proof strategy is to prohibit information flow between active processes. Suppose that $x \in R_p.var \cup W_p.var$ and x is remote to p. We construct E by the following rules.

- (R1): If x is local to q, where $q \in Y2$, then introduce an edge (p, q).
- (R2): If there is a process $w \neq p$ that writes x in H, where $w \in Y2$, then introduce an edge (p, w).

By (C2), p introduces at most one edge. In particular, if x is local to q, then no process $w \neq q$ writes x in H.

Assume that $|Y2| = m$. At most one edge is introduced for each remote event, so m processes in $Y2$ may introduce at most m edges. Let V_0 denote the number of vertices that have no incident edge, let V_1 denote those that have exactly one incident edge, let V_2 denote those that have exactly two incident edges, and let V_3 denote those that have at least three incident edges. By counting the number of vertices, we have $V_0 + V_1 + V_2 + V_3 = m$, and hence

$$V_1 + V_2 + V_3 = m - V_0 \quad . \tag{1}$$

Observe that if the event of p introduces an edge by (R1) or (R2), then the edge is incident to p. This implies that the number of edges is $m - V_0$. Because the sum of the degrees of all vertices is twice the number of edges, we have

$$V_1 + 2V_2 + 3V_3 \leq 2(m - V_0) \quad . \tag{2}$$

Subtracting (1) from (2) yields $V_2 + 2V_3 \leq m - V_0$, which implies that $V_3 \leq \lfloor m/2 \rfloor$ holds. We conclude

that at least $\lceil m/2 \rceil$ vertices have at most two incident edges each.

Eliminate all vertices that have at least three edges and remove those edges from the graph. Then, because $m \geq \lceil (n-1)/w \rceil$, there are at least $\lceil (n-1)/2w \rceil$ vertices that remain. Consider maximal connected subgraphs of the resulting graph. Because each vertex has at most two incident edges, each subgraph is either an acyclic graph or a simple cycle. We now eliminate some vertices (and edges incident to them) if necessary in order to make each connected subgraph bipartite. Note that acyclic graphs and even-length cycles are bipartite. If a subgraph is an odd-length cycle, then it has at least three vertices and the same number of edges by our construction. In each such subgraph, we eliminate one vertex and two incident edges to get an acyclic subgraph, which is bipartite. Thus, we can make every subgraph bipartite by eliminating at most one third of the vertices. Observe that at least $\lceil (n-1)/3w \rceil$ vertices are not eliminated. Vertices in a bipartite graph can be partitioned into two subsets so that each edge joins two vertices in different sets. For each such subgraph, we eliminate vertices in the smaller of the two subsets and the edges incident to them. The resulting graph has at least $\lceil (n-1)/6w \rceil$ vertices and has no edges. These remaining vertices represent the subset of processes selected from the original n processes in Y. We use Z to denote this subset of Y.

We partition Z into two subsets, $Z1$ and $Z2$, so that, for $i \in Z1$, $W_i.var$ contains no remote variable, and for $j \in Z2$, $W_j.var$ contains a remote variable. Informally, $Z1$ is the set of processes that do not write a remote variable, and $Z2$ are those that write a remote variable. Without loss of generality, assume the processes are numbered so that $Z1 = \{1, 2, \cdots, |Z1|\}$, and $Z2 = \{|Z1|+1, |Z1|+2, \cdots, |Z1|+|Z2|\}$. Observe that $|Z1| + |Z2| = |Z|$. The computation G that we seek is defined as follows.

$$G \equiv H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|) \circ \langle [R_1, W_1, 1], [R_2, W_2, 2], \cdots, [R_{|Z|}, W_{|Z|}, |Z|] \rangle$$

To complete the proof, we must show that G is in C and that G satisfies (C1) through (C4). For brevity, we only provide a sketch of these arguments, deferring detailed proofs to the full paper.

To see that $G \in C$, note that by Lemma 1, $H_Z \in C$. Because H satisfies (C2), and because each $L(j)$ consists of only local events, it is straightforward to show that $H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|)$ is also in C. Using this fact, it is possible to show that events of the form $[R_i, W_i, i]$ can be inductively appended to $H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|)$ and the resulting computation is in C.

To conclude, consider conditions (C1) through (C4). (C4) clearly holds by the construction of G. Because H satisfies (C1), (C2), and (C3), and because each $L(i)$ consists only of local events, $H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|)$ satisfies (C1), (C2), and (C3). Thus, it suffices to consider events of the form $[R_i, W_i, i]$. Because events of this form are ordered so that all reads precede all writes, and no two writes access the same variable, by (R2), no such event reads a variable that is written by another process in G. Thus, G satisfies (C1). By (R1), no event $[R_i, W_i, i]$ accesses a local variable of a process that is active in G. Thus, G satisfies (C2). By the definition of $Y2$ and by (R2), for each $j \neq i$, $[R_i, W_i, i]$ does not write a variable that is written by $[R_j, W_j, j]$ or by any event of process j in H_Z or $L(j)$. Thus, G satisfies (C3). This concludes the proof of Lemma 3. □

Theorem 2: For any $S = (C, P, V)$ with write-contention $w > 1$ that solves the minimal mutual exclusion problem, there exists an i-computation in C that contains $\Omega(\log_w N)$ remote events but no Eat_i event.

Proof: $\langle\rangle$ is a P-computation and satisfies (C1), (C2), (C3), and (C4). By repeatedly applying Lemma 3, this implies that there exists a computation F in C that satisfies (C4) and that contains $\Omega(\log_w N)$ remote events of some process i in P. By Lemma 1, $F_i \in C$ holds, from which the theorem follows. □

Corollary 2: For any system S satisfying the conditions of Theorem 2, there exist $\Omega(N)$ processes i in P for which the conclusion of the theorem holds. □

It is possible to show that the bound of Theorem 2 is asymptotically tight for any value of w. In particular, an algorithm by Mellor-Crummey and Scott given in [11] solves the mutual exclusion problem for w processes, in $O(1)$ time, with access-contention (and hence write-contention) w. By applying this solution within a balanced w-ary tree with N leaves, it is possible to obtain an N-process $O(\log_w N)$ mutual exclusion algorithm with access-contention w.

Note that Mellor-Crummey and Scott's algorithm uses load-and-store and compare-and-swap. Even with weaker atomic operations, logarithmic behavior can be achieved. In particular, an N-process $O(\log_2 N)$ mutual exclusion algorithm based on read/write atomicity has been given by Yang and Anderson in [14]. This algorithm has access-contention (and hence write-contention) two.

3.2 Bounds for Cache-Coherent Multiprocessors

On cache-coherent shared memory multiprocessors, the number of remote memory references may be reduced: if a process repeatedly accesses the same remote variable, then the first access may create a copy of the variable in

a local cache line, with further accesses being handled locally. In this section, we count the number of distinct remote variables a process must access to solve the minimal mutual exclusion problem. A lower bound on such a count not only implies a lower bound on the number of cache misses a process causes, but also implies that these cache misses will incur global traffic.

We prove two lower bounds. First, in Theorem 3 below, we show that if the conditions of Theorem 2 are strengthened so that at most c processes can concurrently access (read *or* write) any variable, then some process accesses $\Omega(\log_c N)$ distinct remote variables before eating. Second, in Theorem 4 below, we show that with the conditions of Theorem 2 unchanged, i.e., write-contention is w, then some process accesses $\Omega(\sqrt{\log_w N})$ distinct remote variables before eating. Before establishing the first of these results, we introduce some additional definitions.

Definition: Consider a remote event e of a process p in a computation H. Let x be the remote variable accessed by e. If e is the first event by p in H that accesses x, then we say that e is an *expanding event* in H. If e is a read (write) event, and if e is the first event by p in H that reads (writes) x, then we say that e is an *expanding read (write) event* in H. □

An expanding event is either an expanding read or an expanding write. Note, however, that an expanding read (write) is not necessarily an expanding event. We count the number of expanding events in order to determine the number of distinct remote variables accessed. Observe that if a process executes r expanding events, then it accesses at least r distinct remote variables.

Because the first result of this section is based on a restriction on all concurrent accesses (rather than only concurrent writes) of the same variable, it is necessary to replace condition (C3) by the following.

- (C5) For any events $[R, W, i]$ and $[T, U, j]$ in H, if $((R.var \cup W.var) \cap (T.var \cup U.var)) \neq \{\}$, then $i = j$. Informally, each variable is accessed by at most one process in H.

Our next lemma provides the induction step that leads to the lower bound in Theorem 3.

Lemma 4: Let $S = (C, P, V)$ be a shared memory system with access-contention c that solves the minimal mutual exclusion problem. Let $Y \subseteq P$ be a set of n processes, and let H be a Y-computation in C satisfying (C2), (C4), and (C5) such that each process in Y executes r expanding remote events in H. Then, there exist $Z \subseteq Y$, where $|Z| = \lceil (n-1)/6c \rceil$, and a Z-computation G in C satisfying (C2), (C4), and (C5) such that each process in Z executes $r+1$ expanding remote events in G.

Proof Sketch: The proof is similar to the proof of Lemma 3 and hence is omitted. The central difference between the two proofs lies in the fact that access-contention is restricted here whereas only write-contention is restricted in Lemma 3. In particular, because H satisfies (C5), it is possible to prove a result similar to Lemma 2 showing that at least $n-1$ processes have a "next" expanding remote event after H. The rest of the argument is almost identical to that given in the proof of Lemma 3. □

Theorem 3: For any $S = (C, P, V)$ with access-contention $c > 1$ that solves the minimal mutual exclusion problem, there exists an i-computation in C containing no Eat_i event in which $\Omega(\log_c N)$ distinct remote variables are accessed.

Proof: $\langle\rangle$ is a P-computation and satisfies (C2), (C4), and (C5). By repeatedly applying Lemma 4, this implies that there exists a computation F in C that satisfies condition (C4) and that contains $\Omega(\log_c N)$ expanding remote events of some process i in P. By Lemma 1, $F_i \in C$ holds, from which the theorem follows. □

Corollary 3: For any system S satisfying the conditions of Theorem 3, there exist $\Omega(N)$ processes i in P for which the conclusion of the theorem holds. □

The tree-based algorithms mentioned after Corollary 2 have time complexity $O(\log_c N)$, i.e, the bound of Theorem 3 is asymptotically tight for any value of c.

In the remainder of this section, we prove a lower bound on the number of distinct remote variables required for solving the minimal mutual exclusion problem with write-contention w. Before proving this result, we present some additional definitions.

Definition: Consider a computation H that contains a nonexpanding remote write (read) event e by process i. Let x denote the remote variable accessed by e, and let f be the last remote write (read) of x by process i that precedes e in H. We call f the *predecessor* of e in H. □

Definition: Consider a remote event e of a process i in a computation H. Event e is a *critical* event iff one of the following holds: e is an expanding write; e is an expanding read; e is a nonexpanding event and there is an expanding write by i between e and its predecessor in H. □

The next lemma is a variation of Lemma 2 that deals with critical remote events. Suppose that $S = (C, P, V)$ solves the minimal mutual exclusion problem and let $i \in P$ and $H \in C$. Corresponding to the definition

prior to Lemma 2, we say that *i has a critical remote event after H* iff there exists an i-computation M such that M does not contain Eat_i, M has a critical remote event, and $H \circ M \in C$.

Lemma 5: Suppose that $S = (C, P, V)$ solves the minimal mutual exclusion problem. Let $Y \subseteq P$ be a set of n processes, and let H be a Y-computation in C satisfying (C1), (C2), (C3), and (C4). Then, at least $n-1$ processes in Y have a critical remote event after H.

Proof Sketch: Lemma 2 implies that at least $n-1$ of the processes in Y have a remote event after H. Let p denote such a process and suppose that the next remote event of p is noncritical. Then, there exists a computation $H \circ L \circ \langle e \rangle$ in C, where L is a p-computation consisting of only local events, and e is a noncritical remote event of p. Let x denote the remote variable accessed by e. We show that there exists a computation G in C, obtained by rearranging the events of $H \circ L \circ \langle e \rangle$, that satisfies conditions (C1) through (C4).

Because e is noncritical, H is of the form $X \circ \langle f \rangle \circ Y$, where f is the predecessor of e in $H \circ L \circ \langle e \rangle$, and Y contains no expanding write by p. It follows that each event in Y_p is either a local event of p, a remote read, or a remote write of a variable that is also written by p in $X \circ \langle f \rangle$. From this, it can be shown that $X \circ \langle f \rangle \circ Y_p \circ (Y - Y_p)$ is a computation in C that satisfies (C1), (C2), (C3), and (C4). Because L consists of only local events of p, it is straightforward to show that $X \circ \langle f \rangle \circ Y_p \circ L \circ (Y - Y_p)$ is also a computation in C that satisfies (C1), (C2), (C3), and (C4). Call this last computation G'.

To conclude the construction of G, observe that e and f are both either remote reads of x or remote writes of x. From this, it can be shown that $G \equiv X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle \circ (Y - Y_p)$ is a computation in C. We now show that G satisfies conditions (C1), (C2), (C3), and (C4). If e is a remote read, then f is also a remote read. In this case, either both events read the same value for x in G, or e reads a value written by p in Y_p. In either case, because G' satisfies (C1), G also satisfies (C1). Because G' satisfies (C2), and because e and f access the same remote variable, G also satisfies (C2). If e is a remote write, then f is also a remote write. Hence, because G' satisfies (C3), G also satisfies (C3). Finally, because G' satisfies (C4), and e is a remote event, G satisfies (C4).

To summarize, we have shown that if some process in Y has a next remote event after H that is noncritical, then there exists a Y-computation in C satisfying (C1), (C2), (C3), and (C4) that contains more remote events than H. If this argument could be applied repeatedly, then it would possible to construct a computation in C that violates the Progress requirement. This proves the lemma. □

The next lemma is a stronger version of Lemma 3 in which only critical remote events are counted rather than all remote events.

Lemma 6: Let $S = (C, P, V)$ be a shared memory system with write-contention w that solves the minimal mutual exclusion problem. Let $Y \subseteq P$ be a set of n processes, and let H be a Y-computation in C satisfying (C1), (C2), (C3), and (C4) such that each process in Y executes r critical remote events in H. Then, there exist $Z \subseteq Y$, where $|Z| = \lceil (n-1)/6w \rceil$, and a Z-computation G in C satisfying (C1), (C2), (C3), and (C4) such that each process in Z executes $r+1$ critical remote events in G.

Proof Sketch: The proof is almost identical to that of Lemma 3, except that Lemma 5 is used instead of Lemma 2. □

According to the following theorem, among the $\Omega(\log_w N)$ remote events mentioned in Theorem 2, $\Omega(\sqrt{\log_w N})$ distinct remote variables are accessed.

Theorem 4: For any $S = (C, P, V)$ with write-contention $w > 1$ that solves the minimal mutual exclusion problem, there exists an i-computation in C containing no Eat_i event in which $\Omega(\sqrt{\log_w N})$ distinct remote variables are accessed.

Proof: $\langle\rangle$ is a P-computation and satisfies (C1), (C2), (C3), and (C4). By repeatedly applying Lemma 6, this implies that there exist a computation F in C that satisfies (C4) and that contains $\Omega(\log_w N)$ critical remote events of some process i in P. By Lemma 1, $F_i \in C$. Let D denote the number of expanding events in F_i, let W denote the number of expanding writes in F_i, let R denote the number of expanding reads in F_i, and let E denote the number of nonexpanding critical remote events in F_i. Then,

$$(W + R + E) \geq c \cdot \log_w N \qquad (3)$$

holds for some positive constant c. Observe that D is at least as big as W or R. Also, D is at least as big as the number of distinct remote variables accessed by events in E. A remote variable can be accessed multiple times by events in E only if there is an intervening write from W between any two such accesses. It follows, then, that

$$D \geq \mathbf{max}(W, R, E/(W+1)) \quad . \qquad (4)$$

We now show that $D \geq m \cdot \sqrt{\log_w N}$ for some positive constant m. Assume, to the contrary, that $D < m \cdot \sqrt{\log_w N}$. Then, by (4), we have $W < m \cdot \sqrt{\log_w N}$ and $R < m \cdot \sqrt{\log_w N}$. By (3), this

implies that

$$\frac{E}{W+1} > \frac{c \cdot \log_w N - 2m \cdot \sqrt{\log_w N}}{m \cdot \sqrt{\log_w N} + 1} \quad .$$

By (4), this inequality implies that $D \geq s \cdot \sqrt{\log_w N}$ for some positive constant s. □

Corollary 4: For any system S satisfying the conditions of Theorem 4, there exist $\Omega(N)$ processes i in P for which the conclusion of the theorem holds. □

3.3 Relaxing the Atomicity Assumption

The results we have presented so far are dependent on the Atomicity Assumption given in Section 2. In this section, we show that our main results can be generalized if this assumption is relaxed to allow events to access multiple remote variables.

First, let us consider Theorem 2. The crux of the proof of this theorem is the graph-based argument given in Lemma 3. This argument was used to reduce the original set of n processes to a set of $\lceil (n-1)/6w \rceil$ processes whose next remote event can be applied without violating conditions (C1) through (C4). The argument is based on the assumption that each process (in $Y2$) introduces at most one incident edge.

If each remote event is allowed to access at most v remote variables, then this argument can be generalized as follows. It can be shown that concurrent writes to the same variable (among the remote events yet to be applied to H) can be eliminated in the original set of n processes by reducing to a set of $\lceil (n-1)/vw \rceil$ processes. If the graph argument is then applied to this reduced set of processes, then it can be shown that each process introduces at most v incident edges. We number the edges introduced by each process from 1 up to (at most) v. We then apply the original graph argument to the edges numbered "1". This reduces the number of processes from $\lceil (n-1)/vw \rceil$ to $\lceil (n-1)/6vw \rceil$ and eliminates all edges numbered "1". We apply the same argument again to eliminate the edges numbered "2", and the number of processes is further reduced to $\lceil (n-1)/6^2 vw \rceil$. If this argument is repeated until all edges have been eliminated, then we reduce the number of processes to $\lceil (n-1)/6^v vw \rceil$.

Note that the argument of the previous paragraph eliminates any conflicts between those events that are yet to be applied and previous events (i.e., those in computation H of Lemma 3). However, there might be additional conflicts among the events to be applied. (In Lemma 3, all such events are either remote writes to distinct variables or remote reads. In this case, by applying all reads first, further conflicts can easily be avoided.) Note, however, that the argument of the previous paragraph can be applied yet again to remove all such conflicts. This reduces the number of processes by another factor of $1/6^v$. Putting this all together, we have reduced the original set of n processes to a set of $\lceil (n-1)/6^{2v} vw \rceil$ processes whose next remote event can be applied without violating conditions (C1) through (C4). Using this result as an induction step, it is possible to establish the following theorem, which generalizes Theorem 2.

Theorem 5: For any $S = (C, P, V)$ with write-contention $w > 1$ that solves the minimal mutual exclusion problem, if each event accesses at most v remote variables, then there exists an i-computation in C that contains $\Omega((\log_2 N)/(v + \log_2 w))$ remote events but no Eat_i event. □

Similarly, Theorems 3 and 4 can be generalized as follows.

Theorem 6: For any $S = (C, P, V)$ with access-contention $c > 1$ that solves the minimal mutual exclusion problem, if each event accesses at most v remote variables, then there exists an i-computation in C containing no Eat_i event in which $\Omega((\log_2 N)/(v + \log_2 c))$ distinct remote variables are accessed. □

Theorem 7: For any $S = (C, P, V)$ with write-contention $w > 1$ that solves the minimal mutual exclusion problem, if each event accesses at most v remote variables, then there exists an i-computation in C containing no Eat_i event in which $\Omega(\sqrt{(\log_2 N)/(v + \log_2 w)})$ distinct remote variables are accessed. □

Corollaries similar to those given previously also follow from Theorems 5 through 7. Detailed proofs of these three theorems will be presented in the full paper.

4 Concluding Remarks

The time bounds proved in this paper establish that trade-offs exist between time complexity and write- and access-contention in solutions to the minimal mutual exclusion problem. The results of Section 3.3 show that similar trade-offs exist between time complexity and atomicity. Because any algorithm that solves the leader election or mutual exclusion problems also solves the minimal mutual exclusion problem (this will be shown formally in the full paper), these trade-offs apply to these problems as well.

For wait-free algorithms, Herlihy has characterized synchronization primitives by consensus number [6]. Such a characterization is not applicable when waiting is introduced. One way of determining the power of synchronization primitives in this case is to compare the

time complexity of mutual exclusion using such primitives. For instance, it is possible to solve the mutual exclusion problem with $O(1)$ time complexity using load-and-store or fetch-and-add, while the best-known upper bound for read/write algorithms is $O(\log_2 N)$ [14]. If a lower-bound result could be proved showing that this gap is fundamental, then this would establish that reads and writes are weaker than read-modify-writes from a performance standpoint. This would provide contrasting evidence to Herlihy's hierarchy, from which it follows that reads and writes are weaker than read-modify-writes from a *resiliency* standpoint. It is interesting to note that there exist read/write mutual exclusion algorithms with write-contention N that have $O(1)$ time complexity in the absence of competition [1, 9, 14]. Thus, establishing the above-mentioned lower bound for read/write algorithms will require proof techniques that differ from those given in this paper.

We do not know whether the bound given in Theorem 4 is tight. We conjecture that this bound can be improved to $\Omega(\log_w N)$, which has a matching algorithm [14].

One may be interested in determining the effect of contention on space requirements. It is quite easy to show that solving the minimal mutual exclusion problem with write-contention w requires at least N/w variables. In particular, it can be shown that every process writes a variable before eating. So, consider the computation in which every process is enabled to perform its first write. Because write-contention is w, the total number of variables enabled to be written is $\Omega(N/w)$. It can be shown that this bound is tight; we defer a detailed proof of this result to the full paper.

It is our belief that the most important contribution of this paper is to show that meaningful time bounds can be established for concurrent programming problems for which busy-waiting is inherent. We hope that our work will spark new work on time complexity results for such problems.

Acknowledgements: We would like to thank Gadi Taubenfeld for prompting us to consider the bounds for cache-coherence presented in Section 3.2. We would also like to thank Sanglyul Min and the anonymous referees for their helpful comments on an earlier draft of this paper.

References

[1] R. Alur and G. Taubenfeld, "Results about Fast Mutual Exclusion", *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*, December, 1992, pp. 12-21.

[2] J. Anderson, "A Fine-Grained Solution to the Mutual Exclusion Problem", *Acta Informatica*, Vol. 30, No. 3, 1993, pp. 249-265.

[3] T. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, January, 1990, pp. 6-16.

[4] E. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, Vol. 8, No. 9, 1965, pp. 569.

[5] C. Dwork, M. Herlihy, and O. Waarts, "Contention in Shared Memory Algorithms", *Proceedings of the 25th ACM Symposium on Theory of Computing*, May, 1993, pp. 174-183.

[6] M. Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, 1991, pp. 124-149.

[7] M. Herlihy, B-H. Lim, and N. Shavit, "Low Contention Load Balancing on Large-Scale Multiprocessors", *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, July, 1992, pp. 219-227.

[8] M. Herlihy, N. Shavit, and O. Waarts, "Low Contention Linearizable Counting", *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, October, 1991, pp. 526-535.

[9] L. Lamport, "A Fast Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems*, Vol. 5, No. 1, February, 1987, pp. 1-11.

[10] N. Lynch and N. Shavit, "Timing-Based Mutual Exclusion", *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*, December, 1992, pp. 2-11.

[11] J. Mellor-Crummey and M. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 9, No. 1, February, 1991, pp. 21-65.

[12] M. Merritt and G. Taubenfeld, "Knowledge in Shared Memory Systems", *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, August, 1991, pp. 189-200.

[13] G. Pfister and A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks", *IEEE Transactions on Computers*, Vol. C-34, No. 11, November, 1985, pp. 943-948.

[14] J. Yang and J. Anderson, "Fast, Scalable Synchronization with Minimal Hardware Support", *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, August, 1993, pp. 171-182.