Jennifer Welch (Ed.)

LNCS 2180

Distributed Computing

15th International Conference, DISC 2001
Lisbon, Portugal, October 2001
Proceedings

Springer

Jennifer Welch (Ed.)

Distributed Computing

15th International Conference, DISC 2001
Lisbon, Portugal, October 3-5, 2001
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Jennifer Welch
Texas A&M University, Department of Computer Science
College Station, TX 77843-3112, USA
E-mail: welch@cs.tamu.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Distributed computing : 15th international conference ; proceedings / DISC 2001, Lisbon, Portugal, October 3 - 5, 2001. Jennifer Welch (ed.). – Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2180)
ISBN 3-540-42605-1

CR Subject Classification (1998): C.2.4, C.2.2, F.2.2, D.1.3, F.1, D.4.4-5

ISSN 0302-9743
ISBN 3-540-42605-1 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10840494 06/3142 5 4 3 2 1 0

Preface

DISC, the International Symposium on DIStributed Computing, is an annual forum for research presentations on all facets of distributed computing. DISC 2001 was held on Oct 3–5, 2001, in Lisbon, Portugal. This volume includes 23 contributed papers. It is expected that these papers will be submitted in more polished form to fully refereed scientific journals. The extended abstracts of this year's invited lectures, by Gerard LeLann and David Peleg, will appear in next year's proceedings.

We received 70 regular submissions. These submissions were read and evaluated by the program committee, with the help of external reviewers when needed. Overall, the quality of the submissions was excellent, and we were unable to accept many deserving papers.

This year's *Best Student Paper* award goes to Yong-Jik Kim for the paper "A Time Complexity Bound for Adaptive Mutual Exclusion" by Yong-Jik Kim and James H. Anderson.

October 2001 Jennifer Welch

Organizing Committee

Chair:	Luis Rodrigues (University of Lisbon)
Publicity:	Paulo Veríssimo (University of Lisbon)
Treasurer:	Filipe Araújo (University of Lisbon)
Web:	Alexandre Pinto (University of Lisbon)
Registration:	Hugo Miranda (University of Lisbon)

Steering Committee

Faith Fich (U. of Toronto)
Maurice Herlihy (Brown U.)
Prasad Jayanti (Dartmouth)
Shay Kutten (Technion)
Michel Raynal (vice-chair) (IRISA)
André Schiper (chair) (EPF Lausanne)
Jennifer Welch (Texas A&M U.)

Program Committee

Marcos K. Aguilera (Compaq SRC)
Mark Moir (Sun Microsystems Laboratories)
Lorenzo Alvisi (U. Texas, Austin)
Stephane Perennes (CNRS U. de Nice INRIA)
Hagit Attiya (Technion)
Benny Pinkas (STAR Lab, Intertrust Technologies)
Shlomi Dolev (Ben-Gurion U.)
Ulrich Schmid (Technical U., Vienna)
Tamar Eilam (IBM T.J. Watson Research Center)
Philippas Tsigas (Chalmers U.)
Amr El-Abbadi (U. California, Santa Barbara)
Jennifer Welch (chair) (Texas A&M U.)
Panagiota Fatourou (Max-Planck Inst. Informatik)
Pierre Wolper (U. Liege)
John Mellor-Crummey (Rice U.)

Outside Referees

Author Index

Table of Contents

A Time Complexity Bound for Adaptive Mutual Exclusion*

(Extended Abstract)

Yong-Jik Kim and James H. Anderson

Department of Computer Science
University of North Carolina at Chapel Hill

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

1 Introduction

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

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

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

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

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

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

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

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

2 Definitions

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

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

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

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

We use $\langle e, \ldots \rangle$ to denote a computation that begins with the event e, and $\langle\rangle$ to denote the empty computation. We use $H \circ G$ to denote the computation obtained by concatenating computations H and G. The value of variable v at the end of computation H, denoted $\mathit{value}(v, H)$, 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 $\mathit{writer\_event}(v, H)$, and its owner is denoted $\mathit{writer}(v, H)$. (Although our definition of an event allows multiple instances of the same event, we assume that such instances are distinguishable from each other.) If v is not written by any event in H, then we let $\mathit{writer}(v, H) = \bot$ and $\mathit{writer\_event}(v, H) = \bot$.

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

The following properties apply to any shared-memory system.

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

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

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

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

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

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

$$\begin{aligned} &H \circ \langle \mathit{Enter}_p \rangle \in C \ \text{ iff } \ \mathit{value}(\mathit{stat}_p, H) = \mathit{ncs}; \\ &H \circ \langle \mathit{CS}_p \rangle \in C \ \text{ only if } \ \mathit{value}(\mathit{stat}_p, H) = \mathit{entry}; \\ &H \circ \langle \mathit{Exit}_p \rangle \in C \ \text{ only if } \ \mathit{value}(\mathit{stat}_p, H) = \mathit{exit}. \end{aligned}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3 Proof Strategy

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

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

$$\begin{aligned} \mathrm{Act}(H) &= \{p \in P \mid H\,|\,p \neq \langle\rangle \textit{ and } \langle Exit_p \rangle \textit{ is not in } H\} \\ \mathrm{Fin}(H) &= \{p \in P \mid H\,|\,p \neq \langle\rangle \textit{ and } \langle Exit_p \rangle \textit{ is in } H\} \end{aligned} \qquad \Box$$

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

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

- **(R1)** *For any event e_p and f_q in H, where $p \neq q$, if p writes to a variable v, and if another process q reads* that *value from v, then $p \in$* Fin(H).
- **(R2)** *If a process p accesses a variable that is local to another process q, then $q \notin$* Act(H).
- **(R3)** *For any variable v, if v is accessed by more than one processes in* Act(H)*, then either $writer(v, H) = \perp$ or $writer(v, H) \in$* Fin(H) *holds.*

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

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

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

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

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

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

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

Since each process in Y' accesses a distinct remote variable in its future critical event, it is clear that each process generates at most one edge by rule (i) and at most $j+1$ edges by rule (ii). By applying Turán's theorem (Theorem 1), we can find a subset Z of Y' such that $|Z| = \Omega(\sqrt{n}/j)$ and their critical events