Computing with Infinitely Many Processes
under assumptions on concurrency and participation

Michael Merritt* Gadi Taubenfeld†

May 25, 2001

Abstract

We explore four classic problems in concurrent computing (election, mutual exclusion, consensus, and naming) when the number of processes which may participate is infinite. Partial information about the number of actually participating processes and the concurrency level is shown to affect the possibility and complexity of solving these problems. We survey and generalize work carried out in models with finite bounds on the number of processes, and prove several new results. These include and improved bound for election when participation is required (even for finitely many processes) and a new adaptive algorithm for starvation-free mutual exclusion in a model with unbounded concurrency. We also explore models where objects stronger than atomic registers, such as test&set bits, semaphores or read-modify-write registers, are used.

*AT&T Labs, 180 Park Ave., Florham Park, NJ 07932-0971. mischu@research.att.com.

†The Open University, 16 Klausner st., P.O.B. 39328, Tel-Aviv 61392, Israel, and AT&T Labs. gadi@cs.openu.ac.il.

1 Introduction

1.1 Motivation

We explore several classic problems in concurrent computing (election, mutual exclusion, and consensus) when the number of processes which may participate is infinite. Partial information about the number of actually participating processes and the concurrency level is shown to affect the possibility and complexity of solving these problems. This paper surveys and generalizes work carried out in models with finite bounds on the number of processes, and proves several new results. These include an improved bound for election when participation is required (even for finitely many processes) and a new adaptive algorithm for starvation-free mutual exclusion when the number of concurrent processes is not bounded *a priori.*

Processes: In most work on the design of shared memory algorithms, it is assumed that the number of processes is finite and *a priori* known. Here we investigate the the design of algorithms assuming no *a priori* bound on the number of processes. In particular, we assume the number of processes may be infinite. Although, in practice, the number of processes is always finite, algorithms designed for an infinite number of processes (when possible) may scale well: their time complexity may depend on the actual contention and not on the total number of processes.

Starvation-free mutual exclusion has been solved in such a model using two registers and two weak semaphores (see Theorem 5.3) [FP87]. Wait-free solvability of tasks when there is no upper bound on the number of participating processes has also been investigated [GK98], but in this earlier work no run has an infinite number of participating processes. A model in which processes are continuously being created and destroyed (and hence their number is not *a priori* known) has also been considered [MT93]. (An underlying, automata-theoretic formulation of infinite concurrency may be found in e.g. [M$^+$93, Lyn96].)

Concurrency: An important factor in designing algorithms where the number of processes is unknown, is the *concurrency level,* the maximum number of processes that may be active simultaneously. (That is, participating in the algorithm at the same instant of time.[1]) We distinguish between the following concurrency levels:

- *finite:* there is a finite bound (denoted by c) on the maximum number of processes that are simultaneously active, over all runs. (The algorithms in this paper assume that c is known.)
- *bounded:* in each run, there is a finite bound on the maximum number of processes that are simultaneously active.
- *unbounded:* in each run, the number of processes that are simultaneously active is finite but can grow without bound.

[1] We have chosen to define the concurrency level of a given algorithm as the maximum number of processes that can be simultaneously active. A weaker possible definition of concurrency, which is not considered here, is to count the maximum number of processes that actually take steps while some process is active. For example, assume that (1) processes p_1, p_2 and p_3 are active during the time intervals T_1, T_2 and T_3, respectively, and (2) T_1 overlaps with both T_2 and T_3, but T_2 and T_3 do not overlap. Then the concurrency level, as we define it, is 2, while according to the weaker definition it is 3.

- *infinite:* the maximum number of processes that are simultaneously active may be infinite.

The case of infinite concurrency raises some issues as to the appropriate model of computation and semantics of shared objects. We assume that an execution consists of a (possibly infinite) sequence of *group steps*, where each group step is itself a (possibly infinite) sequence of steps, but containing at most *one* primitive step by each process. Hence, no primitive step has infinitely many preceding steps by any single process. The semantics of shared objects need to be extended (using limits) to take into account behavior after an infinite group step. This leaves some ambiguity when a natural limit does not exist. For example, what value will be returned by a read which follows a step in which infinitely many processes write different values? Natural limits exist for all the executions we consider. In particular, in the algorithm in Figure 5, the only write steps assign the value 1 to shared bits (which have initial value 0). The well-ordering of primitive steps in an execution of this model assures that there is a first write to any bit, and all subsequent reads will return 1.

Participation: When assuming a fault-free model with *required participation* (where every process has to start running at some point), many problems are solvable using only constant space. However, a more interesting and practical situation is when participation is not required, as is usually assumed when solving resource allocation problems. For example, in the mutual exclusion problem a process can stay in the reminder region forever and is not required to try to enter its critical section.

We use the notation $[\ell, u]$-*participation* to mean that at least ℓ and at most u processes participate. Thus, for a total of n processes (where n might be infinite) $[1, n]$-participation is the same as saying that participation is not required, while $[n, n]$-participation is the same as saying that participation is required. Requiring that all processes must participate does not mean that there must be a point at which they all participate at the same time. That is, the concurrency level might be smaller that the upper bound on the number participating processes. Notice also that if an algorithm is correct assuming $[\ell, u]$-participation, then it is also correct assuming $[\ell', u']$-participation, where $\ell \leq \ell' \leq u' \leq u$.

Thus, any solution assuming that participation is not required, is correct also for the case when participation is required, and hence it is expected that such solutions (for the case where participation is not required) may be less efficient and harder to construct.

1.2 Properties

We define two properties of algorithms considered in the paper.

Adaptive algorithms: An algorithm is *adaptive* if the time complexity of processes' operations is bounded by a function of the actual concurrency. The term *contention sensitive* was first used to describe such algorithms [MT93], but later the term *adaptive* become commonly used. Time complexity is computed using the standard model, in which each primitive operation on an objects is assumed to take no more than one time unit. In the case of mutual exclusion algorithms, we measure the maximum time between releasing the critical section, until the critical section is re-entered. For models in which there is a minimum bound on the number of participating processes, we measure one time unit to the first point at which the minimum number of processes have begun executing the algorithm.

Symmetric algorithms: An algorithm is *symmetric* if the only way for distinguishing processes is by comparing (unique) process identifiers. Process id's can be written, read, and compared, but there is no way of looking inside any identifier. Thus, identifiers cannot be used to index shared registers. Various variants of symmetric algorithms can be defined depending on how much information can be derived from the comparison of two unequal identifiers. In this paper we assume that id's can only be compared for equality. (In particular, there is no total order on process id's in symmetric algorithms.)[2]

1.3 Summary of results

We assume the reader is familiar with the definitions of the following problems:

1. The mutual exclusion problem, which is to design a protocol that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65];

2. The consensus problem, which is to design a protocol in which all correct processes reach a common decision based on their initial opinions [FLP85];

3. The (leader) election problem, which is to design a protocol where any fair run has a finite prefix in which all processes (correct or faulty) commit to some value in $\{0, 1\}$, and exactly one process (the leader) commits to 1; We do not require the leader to be identified to the other processes, although we do require all processes to terminate in fair runs. In a model with infinitely many processes, identifying the leader obviously requires infinite space–this weaker assumption makes the lower bounds more complex. To prevent trivial solutions, it is assumed that the ids are not initially known to the processes, although we do assume that process identities are natural numbers.

4. The *wait-free naming* problem which is to assign unique names to initially identical processes. Every participating process is able to get a unique name in a finite number of steps regardless of the behavior of other processes.

We show that even with a fixed bound on concurrency and required participation, election (using registers) requires infinite shared space. Among the novel algorithms presented are two demonstrating that either a single shared register of infinite size, or infinitely many shared bits, suffice for both election and consensus. (In addition, the first algorithm is adaptive.) If in addition test&set bits are used, then solving the above problem requires only finite space; however, a result of Peterson ([Pet94]) implies that the more complex problem of starvation-free mutual exclusion (bounded concurrency, participation not required) still requires infinite space. In fact, even using read-modify-write registers to solve this problem, a result of Fischer *et al* ([F$^+$89]) implies that infinite space is required. However, Friedberg and Peterson ([FP87]) have shown that using objects such as semaphores that allow waiting enables a solution with only constant space.

When there is no upper bound on the concurrency level, we show that even infinitely many test&set bits not suffice to solve naming. Naming can be solved assuming bounded

[2] Styer and Peterson, [SP89], discuss two types of symmetry. Our notion corresponds to their stronger restriction, "symmetry with equality". Some of our asymmetric algorithms, presented later, work in systems with their weaker symmetry restriction, "symmetry with arbitrary comparisons," in that they depend upon a total order of the process id's, but do not (for example) use id's to index arrays.

concurrency using test&set bits only, hence it separates bounded from unbounded concurrency. At the boundary of unbounded and infinite concurrency, Friedberg and Peterson ([FP87]) have shown a separation for starvation-free mutual exclusion using registers and weak semaphores: two of each suffice for unbounded concurrency, but the problem cannot be solved using these primitives for infinite concurrency.

The tables below summarize the results discussed in this paper. As indicated, many are derived from existing results, generally proven for models where the number of processes is finite and known. We use the following abbreviations: DF for deadlock-free; SF for starvation-free; mutex for mutual exclusion; U for Upper bound; L for Lower bound; RW for atomic read/write registers; T&S for test&set bits; wPV for weak semaphores; and RMW for read-modify-write registers. (The default is "No" for the adaptive and symmetric columns. All lower bounds hold for non-adaptive and asymmetric case.)

Problem			Model		Result					
	Name	Bound	Concurrency $c > 1$	Participation	Properties: adaptive?	Properties: symmetric?	Space: #	Space: size	Thm #	Using results from
1	Election	L_1	c	$[c, \infty]$			∞ state space		2.1	
		U	c	$[c, \infty]$			∞	2	2.2	
	(Upper	U	2	$[2, \infty]$	Y	Y	1	∞	2.3	
	bounds	U	c	$[c, \infty]$	Y	Y	2	∞	2.4	
	hold	U	c	$[c, \infty]$	Y		1	∞	2.5	
	also for	U	c	$[\ell, \infty]$ $\ell < c$			$\log(c - \ell + 1) + 2$	∞	3.2	
	consen-	U	c	$[1, \infty]$		Y	$\log c + 1$	∞	3.1	[SP89]
	sus.)	L_2	c	$[1, \infty]$			$\log c + 1$	2	3.1	[SP89]
2	DF	U	c	$[1, \infty]$		Y	c	∞	3.1	[SP89]
	mutex	L	c	$[1, \infty]$			c	2	3.1	[BL93]
3	Election	L	bounded	$[1, \infty]$			∞	2	3.1	$1L_2$,2L
	SF	U_1	unbounded	$[1, \infty]$	Y	Y	∞	∞	3.3	
	mutex	U_2	infinite	$[1, \infty]$			∞	2	3.4	
4	Test	L	bounded	$[1, \infty]$			∞	2	4.1	3L,5U
	&	U	unbounded	$[1, \infty]$	Y	Y	∞	∞	4.1	$3U_1$
	Set	U	infinite	$[1, \infty]$			∞	2	4.1	$3U_2$

Results using atomic registers in a fault-free model

Problem			Model				Result					
	Name	Bound	Concurrency $c > 1$	Participation	Fault tolerance	Shared objects	Properties: adaptive?	Properties: symmetric?	Space: #	Space: size	Thm #	Using results from
5	Election DF mutex	L	c	$[\infty, \infty]$	0	T&S			1	2		trivial
		U	infinite	$[1, \infty]$	∞	T&S	Y	Y	1	2		trivial
6	SF mutex	L	bounded	$[1, \infty]$	0	RW,T&S			∞	2	4.2	[Pet94]
		U	infinite	$[1, \infty]$	0	RW,T&S			∞	2	3.4	$3U_2$
7	Naming	L&U	bounded	$[1, \infty]$	∞	T&S	Y	Y	∞	2	4.3	
		L	unbounded	$[1, \infty]$	∞	T&S			no alg		4.3	
8	Consensus	L	2	$[\infty, \infty]$	1	RMW			2	2		trivial
		U	infinite	$[1, \infty]$	∞	RMW	Y	Y	1	3		trivial
		U	infinite	$[\infty, \infty]$	1	RMW	Y		4	2	5.1	[LA87]
9	SF mutex	L	bounded	$[1, \infty]$	0	RMW			1	∞	5.2	[F$^+$89]
		U	unbounded	$[1, \infty]$	0	RMW	Y	Y	1	∞	5.2	[B$^+$82]
		U	infinite	$[1, \infty]$	0	RMW			∞	2		$3U_2$
10	SF mutex	U	unbounded	$[1, \infty]$	0	RW,wPV	Y	Y	2,2	2	5.3	[FP87]
		L	infinite	$[1, \infty]$	0	RW,wPV			no alg		5.3	[FP87]

Results using stronger shared objects

2 Atomic registers: participation is required

This section demonstrates that requiring a minimum number of process to participate (the required participation model) is a powerful enabling assumption: the problems we study are solvable with a small number of registers. However, reducing the number of registers does not mean that it is also possible to achieve a finite state space. We begin by showing that any solution to election in this model requires infinite state space. Then we show that election can be solved either by using an infinite number of registers of finite size or a (small) finite number of registers of infinite size.

2.1 A lower bound for election when the concurrency is $c > 1$

Even when participation is required, (and equal to the concurrency) any solution to election for infinitely many processes must use infinite shared space. This holds even if processes do not need to learn the identity of the leader as part of the election algorithm.

Theorem 2.1 *There is no solution to election with finite concurrency $c > 1$ and with $[c, \infty]$-participation using finite shared memory (finitely many registers of finite size).*

Proof: Assume to the contrary that there is such an algorithm using finite space. Consider the solo run $solo(p)$ of each process p, and define $write(p)$ to be the subsequence of write events in $solo(p)$. For each process p, let $repeat(p)$ be the first state of the shared memory that repeats infinitely often in $solo(p)$. (Recall that the algorithm uses finite space by assumption. Also, if $write(p)$ is finite, $repeat(p)$ will necessarily be the state after the last write in $write(p)$.)

Let $\beta(p)$ be the finite prefix of $write(p)$ that precedes the first instance of $repeat(p)$. For each such $\beta(p)$, we construct a *signature* subsequence, $sign(p)$, by removing repetitive states and the intervening steps, as follows: Ignoring the states of the processes for now, let $\beta(p) = x_0, x_1, x_2...$, where the x_j are the successive states of the shared memory in $\beta(p)$. Suppose x_j is the first state that repeats in $\beta(p)$, and that x_k is the last state in $\beta(p)$ such that $x_j = x_k$. Remove the subsequence $x_{j+1}, ..., x_k$ from $\beta(p)$. The resulting sequence $x_0, ..., x_j, x_{k+1}, ...$ is a subsequence of $\beta(b)$ with strictly fewer repeating states. Repeat this step until no state repeats–the resulting sequence is (the signature) $sign(p)$.

Since there are only finitely many states of the shared memory, there exists a single state s and a single sequence γ such that $s = repeat(p)$ and $\gamma = sign(p)$ for infinitely many processes, p. The solo runs of any subset of $C = p_1, ..., p_c$ of these processes are compatible: by scheduling appropriately, we can construct runs in which each of the processes take the same steps as in their solo runs. By an easy case analysis, in one such run no leader is elected, or there are many runs in which two or more leaders are elected, a contradiction.

The (compatible) run of the processes in C is constructed as follows: Let $\gamma = y_1, ..., y_k$. First run these c process until each one of them is about to execute its first write. Run process p_1 until it takes the last step in $\beta(p_1)$ that leaves the memory in state y_1. Repeat for p_2 through p_c, then repeat this loop for each state in γ. The resulting finite run leaves the shared memory in state s, and each of the c processes p_i has taken the steps in $\beta(p_i)$. Since the state s repeats infinitely often in the remainder of each of the solo runs $solo(p_i)$, the run can be extended by using a round-robin schedule to extend by some sequence of steps of each process, always returning the memory to state s.

Notice that the signatures are used only in the first stage of the construction, when the processes run until the memory state is s, in such a way that each process "thinks" it ran solo to get there. After reaching (together) the recurring state s, each process can be run solo until s recurs. Since the same state s recurs infinitely many times for each of the processes in C, we can schedule each in turn that has not terminated to take (some finite, nonzero number) of steps until s occurs next, then schedule the next one (i.e., round robin scheduling). Now everyone either runs forever or terminates as in a solo run. ∎

2.2 Algorithms for election and consensus for concurrency c

Although the previous theorem shows that election for infinitely many processes requires unbounded shared memory, when assuming participation is required, many problems become solvable with a small number of registers. We first study the scenario in which the concurrency is equal to participation (concurrency c and participation $[c, \infty]$). We show that election can be solved either by (1) using an infinite number of atomic bits, or (2) using a single register of infinite size. We also present two simple symmetric algorithms.

Theorem 2.2 *For any finite concurrency c, with $[c, \infty]$-participation, there are non-adaptive asymmetric solutions to election (and consensus) using an infinite number of atomic bits.*

Proof: We identify each process with a unique natural number. (But each natural number is not necessarily the identity of a participating process. We always take 0 as a process id not assigned to any process.) The algorithm uses an infinite array $b[0], b[1], ...$ of bits, which are initially 0.

The first step of process i is to read $b[0]$. If $b[0]$ value is 1, it knows that a leader has already been elected and terminates. Otherwise, process i sets $b[i]$ to 1. Then, process i

```
Process i's program
Shared:
        (Leader, Marked): (Process id, boolean) initially (0, 0)
Local:
        local_leader: Process id

if Marked = 0 then
    (Leader, Marked) := (i, 0)
    await (Leader ≠ i) ∨ (Marked = 1)
    local_leader := Leader
    (Leader, Marked) := (local_leader, 1)
fi
return(Leader)
```

Figure 1: Symmetric election for concurrency 2

scans the array until it notices that c other bits (other than $b[0]$) are set. In order not to miss any bit that is set, it scans the array according to a diagonalization: a schedule that visits each bit an infinite number of times. (A canonical example is to first read $b[1]$; then $b[1], b[2]$; then $b[1], b[2], b[3]$, etc.) Once a process notices that c bits are set, it set $b[0]$ to 1. The process with the smallest id among the c that are set to 1 is the leader. By scanning the bits after reading $b[0]$ as 1, the other processes can also learn the id of the leader. This solution for election can be trivially modified, using one other bit, to solve consensus. ∎

A symmetric election algorithm is presented in [SP89], for n processes (with concurrency level n), which use only three atomic registers. Below we present three election algorithms: a symmetric algorithm for concurrency 2 using one register, a symmetric algorithm for concurrency c using two registers, and finally, an asymmetric algorithm for concurrency c using one register.

Theorem 2.3 *For finite concurrency $c = 2$ with $[2, \infty]$-participation, there are adaptive symmetric solutions to election and consensus using one atomic register.*

Proof: The algorithm in Figure 1 is very simple. The register value contains two fields, $Leader$ and $Marked$, respectively, where $Leader$ is a process id (or 0) and $Marked$ is boolean. The statement **await** *condition* is used as an abbreviation for **while** $\neg$*condition* **do** *skip*, and hence, may involve many accesses to the shared memory.

In this very simple algorithm the second process to write to the shared register is elected. The first process to write spins until the second process writes, then marks the second process as the winner. The concurrency bound guarantees no other process will interfere.

This election algorithm can be easily converted into a consensus algorithm by appending the input value (0 or 1) to each process id. The input value of the leader is the consensus value. ∎

The next algorithm, for concurrency $c \geq 2$, is similar to the previous, in that the last participant to write the $Leader$ field is then $Marked$ as the leader. A second register,

```
Process i's program.
Shared:
        (Leader, Marked): (Process id, boolean), initially (0, 0)
        Union: set of at most c process ids, initially ∅
Local:
        local_leader: Process id
        local_union1: set of at most c process ids
        local_union2: set of at most c process ids, initially {i}

if Marked = 0 then (Leader, Marked) := (i, 0) fi
local_union1 := Union
while |local_union1| < c do
    if ¬(local_union2 ⊆ local_union1) then
        Union := local_union1 ∪ local_union2
    fi
    local_union2 := local_union1 ∪ local_union2
    local_union1 := Union
od
local_leader := Leader
(Leader, Marked) := (local_leader, 1)
return(Leader)
```

Figure 2: Symmetric election for concurrency c

$Union$, is used by the first c participants to ensure that they have all finished writing their id into $Leader$. (Note, that in doing so, they learn each other's id's.) Because c is also the number of required participants, each of the first c participants can spin on the $Union$ register until the number of id's in it is c.

Theorem 2.4 *For any finite concurrency c with $[c, \infty]$-participation, there are adaptive symmetric solutions to election and consensus using two atomic registers.*

Proof: In the algorithm in Figure 2, the first register, $Union$, holds a set of at most c processes ids; the second register contains a process id (in the $Leader$ field) and a bit (the $Marked$ field). We first prove that all the processes that terminate elect exactly the same participant as a leader. First, only participating process id's can appear in the shared $Union$ register. A process sets $Marked$ from 0 to 1 and terminates only after it determines that c processes are already participating. Thus, the first c processes to participate will first see that $Marked$ is not set, and no later process can participate until one of these terminates, after which $Marked$ is set. Hence, only the first c processes will write their id's into $Leader$. After this happens, some process may notice that c processes are already participating and set $Marked$ to 1. This implies that only the last process to write $Leader$ can be elected.

To prove termination we observe the following: (1) The first c processes cannot read-spin on a set in $Union$ smaller than c forever: one will read a set in which (at least) itself is absent. (2) Every write to $Union$ is preceded by a read of a set containing fewer values. Suppose there are infinitely many writes. Then let $c' < c$ be the smallest value such that sets of size c' are written infinitely often. It follows that there are infinitely many reads (and hence writes) of sets with fewer than c' elements, a contradiction.

Thus, at some point a process will learn that c processes are participating (i.e., $|Union| = c$) and set *Marked* to 1, allowing the rest of the processing to terminate. This completes the correctness proof of the election algorithm.

As above, this election algorithm can be easily converted into a consensus algorithm, by adding the input value (0 or 1) to each process id. ∎

Finally, we present an asymmetric solution to election (and consensus) using a single atomic register. Asymmetric algorithms allow comparisons between process id's (we assume a total order), however, since the identities of the participating process are not initially known to all processes, the trivial solution where all processes choose process 1 as a leader is excluded.

Theorem 2.5 *For any finite concurrency c and $[c, \infty]$-participation there is an adaptive asymmetric solution to election and consensus using one atomic register.*

Proof: The algorithm in Figure 3 is an adaptation of the two register algorithm for the symmetric case (Theorem 2.4), where only a single register is used. As before, each of the first c processes registers itself in the shared register and repeatedly updates *Union* with any new information it has obtained (exactly as is done in the repeat loop of the symmetric algorithm). The elected process is the process with the minimum id among the first c processes to participate in the algorithm.

Once a process p observes the size of the set in *Union* to be at least c (we argue below exactly c), it knows the identity of the leader (the minimum in this set). If p is not in the set (is not one of the first c participants) it simply terminates. Otherwise, p performs a handshake with the leader, and terminates once it observes that the $c-1$ handshakes have completed. This is done using a counter, *Sig*, and a bit, *Ack*, to enforce the handshake. Process p waits until the leader sets *Sig* to p's rank among the c initial processes, and then sets *Ack* to 0 acknowledge the signal. The leader itself initiates each handshake. Slow processes may overwrite handshakes in progress, which the leader will detect and re-write. We argue below that this overwriting stops once all the initial c processes exit the while loop in line 2.

Notice that no process exits the while loop in line 2 until c processes have entered the loop. Hence, no other process can take a step until one of these c processes terminates. In turn, none of thes c processes can terminate until $c-1$ handshakes are completed, at which point all of the c processes have read c different values in the *Union* field of the register. Hence, at the point that the first process terminates, *Union* holds c different id's (exactly those of the initial c participants), and none of the initial participants will ever write a different value to *Union*. It follows that any later process will see these same c values in *Union* in line 1, exit the while loop in line 2 immediately, and terminate with the correct leader in line 10. Moreover, the value of *Union* is stable after this point. It follows that any terminating processes will choose the same leader.

To argue that every process terminates, the main difficulty is arguing that the first c processes terminate. As in the two-register algorithm, in any run of the algorithm, the value of the *Union* is eventually stable, containing c values, which we argued above are exactly the id's of the first c participants. This means that every process but the leader stops writing unless it is responding to the appropriate handshake value. By a simple induction on the

```
Process i's program
Shared:  (Union, Sig, Ack): Union a set of at most c process id's,
                0 ≤ Sig ≤ c, Ack ∈ {0, 1}, initially (∅, 0, 0)
Local:   local_union1: set of at most c process id's
         local_union2: set of at most c process id's, initially {i}
         myrank, local_sig1, local_sig2: integers between 0 and c, initially 0
         local_ack1: Boolean, initially 0
         leader: process id

(local_union1, local_sig1, local_ack1) := (Union, Sig, Ack)
while |local_union1| < c do
    if ¬(local_union2 ⊆ local_union1) then
        (Union, Sig, Ack) := (local_union1 ∪ local_union2, 0, 0)
    fi
    local_union2 := local_union1 ∪ local_union2
    (local_union1, local_sig1, local_ack1) := (Union, Sig, Ack)
od
local_union2 := local_union1
leader := min{q : q ∈ local_union1}
if i ∉ local_union1 then
    return(leader)
elseif i ≠ leader then
    myrank := |{h ∈ local_union1 : h < i}|
    while local_sig1 < c do
        if (|local_union1| < c) then (Union, Sig, Ack) := (local_union2, 0, 0) /* Register is stale. */
        elseif (local_sig1 = myrank) ∧ (local_ack1 = 1) then
            (Union, Sig, Ack) :=(local_union2, myrank, 0) /* Acknowledge signal. */
        fi
        (local_union1, local_sig1, local_ack1) :=(Union, Sig, Ack)
    od
    return(leader)
else                                              /* |local_union2| = c, i = leader */
    while local_sig1 < c do
        if (|local_union1| < c) ∨ (local_sig1 < local_sig2) then
            (Union, Sig, Ack) := (local_union2, local_sig2, local_ack2)/* Register is stale. */
        elseif local_ack1 = 0 then                     /* Signal acknowledged */
            local_sig2 := local_sig2 + 1
            (Union, Sig, Ack) := (local_union2, local_sig2, 1) /* Send new signal. */
        fi
        (local_union1, local_sig1, local_ack1) := (Union, Sig, Ack)
    od
    return(i)
fi
```

Figure 3: (Asymmetric) election for concurrency c

rank r of the other participants, once each process has read $Union$ to have c values, the process with rank r eventually reads its signal from the leader, the leader eventually reads its acknowledgement, and the process with rank r never executes another write operation. Hence, the leader will eventually increment the Sig field to c and terminate. The other of the first c participants will eventually read $Sig = c$ and terminate, and as we argued above, the later processes will all see c different values in $Union$ and terminate immediately. ∎

3 Atomic registers: participation is not required

As mentioned above, any problem solution in a model in which participation is not required, is also correct when participation is required, and hence such solutions are more difficult to construct.

3.1 Consensus, election and mutual exclusion for concurrency c

We next consider the number and size of registers required to solve consensus, election and mutual exclusion in a model in which participation is not required (i.e., $[1, \infty]$-participation).

Theorem 3.1 *For concurrency level c, the number of atomic registers that are:*
(1) necessary and sufficient for solving deadlock-free mutual exclusion, is c;
(2) necessary and sufficient for solving election, is $\log c + 1$*;*
(3) sufficient for solving consensus, is $\log c + 1$*.*

Proof: (1) The necessary condition follows from a lower bound which was proved in [BL93], and shows that any deadlock-free mutual exclusion algorithm for n processes must use at least n shared registers. The sufficiency condition follows from the existence of a symmetric deadlock-free mutual exclusion algorithm for n processes which uses only n shared registers. This algorithm is presented in [SP89], and it works correctly also for an infinite number of processes provided that the concurrency level is no more than n. (2) The necessary condition follows from a lower bound in [SP89], which shows that any election algorithm for n processes must use at least $\log n+1$ shared registers. The sufficiency condition follows from the existence of a symmetric election algorithm for n processes which uses only $\log n+1$ shared registers. This algorithm is presented in [SP89], and it works correctly also for an infinite number of processes provided that the concurrency level is no more than n. (3) The election algorithm in [SP89] can be easily modified to solve consensus with the same number of registers. Each process when writing its id appends to it its vote (0 or 1). Once a leader is elected, everybody decides on the leader's vote as the consensus value. ∎

Theorem 3.1 implies that when the concurrency level is not finite, an infinite number of registers are necessary for solving the election and mutual exclusion problems. But are an infinite number of registers sufficient? (We observe, based on a result from [YA94], that if we restrict the number of processes that can try to write the same register at the same time, the answer is no.)

3.2 Relating participation and concurrency

The previous section focused on the case in which the number of required participants, ℓ, is equal to the concurrency, c. Note that if $c < \ell$, the notion of required participation depends on termination of at least some of the first c participants. Indeed, the algorithms of the previous section will work correctly for this case, as no process terminates until c have begun, and the leader is determined. If $\ell < c$, the situation is more complex. The following theorem unifies the results of Theorems 2.5 (participation is required) and 3.1 (participation is not required), demonstrating a relation between participation and concurrency in this case.

Theorem 3.2 *For concurrency c with $[\ell, \infty]$-participation where $l < c$,* $\log(c - \ell + 1) + 2$ *registers are sufficient for solving election.*

Proof: The algorithm is a combination of the a single register assuming $[c, \infty]$-participation (from Theorem 2.5), and of the one using $\log c + 1$ registers assuming $[1, \infty]$-participation (from Theorem 3.1). First, we use the single register algorithm (Theorem 2.5) as a filter. Here, instead of choosing a single leader, up to $c - \ell + 1$ processes may be elected. These processes continue to the next level. To implement this, we slightly modify the single register algorithm. A process, needs to wait only until it notices that ℓ (instead of c) processes are participating, and if it is the biggest among them it continues to the next level. Thus, at most $c - \ell + 1$ (and at least one) processes continue to the next level. In that level, they compete using the algorithm of Theorem 3.1 (no change is needed in that algorithm), until one of them is elected. This level requires $\log(c - \ell + 1) + 1$ registers. ∎

3.3 Starvation-free algorithms for unbounded concurrency

Theorem 3.1 leads naturally to the question of whether an infinite number of registers suffice for solving mutual exclusion with unbounded concurrency, when participation is not required.

We present two algorithms that answer this question affirmatively. The first is an adaptive and symmetric algorithm using infinitely many infinite-sized registers. The second is neither adaptive nor symmetric, but uses only (infinitely many) bits.

Theorem 3.3 *There is an adaptive symmetric solution to election, consensus and starvation-free mutual exclusion for unbounded concurrency using an infinite number of registers.*

Proof: These problems can all be solved by simple adaptations to the deadlock-free mutual exclusion algorithm below. This algorithm has three interesting properties: First, it works assuming an unbounded concurrency level, second it is adaptive – its time complexity is a function of the actual number of contending processes, and finally, it is symmetric. Except for the non-adaptive algorithm in the next subsection, we know of no mutual exclusion algorithm (using atomic registers) satisfying the first property. The algorithm is defined formally in Figure 4. In this algorithm, the processes compete in levels, each of which is used to eliminate at least one competing process, until only one process remains. The winner enters its critical section, and in its exit code it publishes the index to the next empty level, so that each process can join the competition starting from that level.

```
Process i's program
Shared:
    next: integer, initially 0
    x[0..∞]: array of integers (the initial values are immaterial)
    b[0..∞], y[0..∞], z[0..∞]: array of boolean, initially all 0
Local:
    level: integer, initially 0
    win: boolean, initially 0

1 start: level := next
2    repeat
3        x[level] := i
4        if y[level]      then  b[level] := 1
5                               await level < next
6                               goto start fi
7        y[level] := 1
8        if x[level] ≠ i then   await (b[level] = 1) ∨ (z[level] = 1)
9                               if z[level] = 1 then await level < next
10                                                   goto start
11                              else level := level + 1 fi
12       else   z[level] := 1
13              if b[level] = 0 then win := 1
14              else level := level + 1 fi fi
15   until win = 1
16   critical section
17   next := level + 1
```

Figure 4: Adaptive deadlock-free mutual exclusion for unbounded concurrency

In line 1, process i learns the level at which it should start competing. In line 3, process i sets x to i so that it can later observe if there is contention. In lines 4-6, if process i notices that $y[level]$ is 1, it knows there is contention. In that case, process i gives up, sets $b[level]$ indicating that some process has lost at that level, and waits for the critical section to be released. Otherwise, in Line 7, process i sets $y[level]$ so that newcomers are blocked. Thus, after line 7, process i has to compete only with those processes which also noticed that $y[level]$ is on. Among these processes only the last to write into x has a chance to win at that level and enter its critical section. Furthermore, if no process is blocked at that level (i.e., set $b[level]$ to 1), then some process must win at that level and is allowed to enter its critical section.

If no process succeeds at the test in line 8, then all the processes that reach line 8, will move one level up, leaving at least one process behind (waiting in line 5). Otherwise, when some process (called the potential winner) succeeds at the test in line 8, there may be a race between the potential winner, which tries to set $z[next]$ first (line 12), and possibly another process that tries to set $b[next]$ (line 5).

Once $z[next]$ is set the potential winner checks the bit $b[next]$. If it is off the potential winner wins the race and can enter its critical section, otherwise it continues to the next level. All other processes wait to see the result of the race (line 8). If they are certain that $b[next]$ was set first, they continue to the next level, otherwise they give up and wait for the critical section to be released.

The adaptive deadlock-free algorithm above is easily modified using standard "helping" techniques to satisfy starvation freedom. Because the number of processes is infinite, a global diagonalization is necessary instead of a round-robin schedule: a process helps others in the order given by an enumeration in which every process id appears infinitely often. That is, processes set flags when they leave the remainder section, and before leaving the critical section, a process examines the flag of the next process in the diagonalization and grants the critical section if it determines the associated process is waiting. (Global variables recording progress in this globalization are maintained via the mutual exclusion of the critical section.) Starvation-freedom follows if each process appears infinitely often in the diagonalization. Even simpler, standard modifications convert the above algorithm to solve leader election or consensus. ∎

The question of designing an adaptive mutual exclusion algorithm, was first raised in [MT93]. The solution presented is efficient provided process creation and deletions are rare (the term *contention sensitive* was suggested but the term *adaptive* become commonly used). In [CS94], the only previously known adaptive mutual exclusion algorithm was presented, in a model where it is assumed that the number of processes (and hence concurrency) is finite. The algorithm exploits this assumption to work in bounded space. The algorithm does not work assuming unbounded concurrency. In [Lam87], a fast algorithm is presented which provides fast access in the absence of contention. However, in the presence of any contention, the winning process may have to check the status of all other n processes (i.e. access n different shared registers) before it is allowed to enter its critical section. A symmetric (non-adaptive) mutual exclusion algorithm for n processes is presented in [SP89].

Theorem 3.4 *There is a non-adaptive asymmetric solution to election, consensus and starvation-free mutual exclusion for infinite concurrency using an infinite number of bits.*

Proof: Figure 5 presents a simple algorithm for election. Modification to solve consensus is trival–starvation-free mutual exclusion can be achieved using techniques similar to those in the previous algorithm. There are four bits associated with each process i, $RaceOwner[i]$, $RaceOther[i]$, $Win[i]$, and $Lose[i]$. The idea is that each process i uses the two bits $RaceOwner[i]$ and $RaceOther[i]$ to race against the other processes. The minimum winner is the leader. The other two bits are use by the owner to signal whether it won or lost. The leader is the minimum i such that $Win[i]$ is set.

Termination: First, a simple argument shows each execution of the wait loop in line 6 eventually terminates. The first process i to perform the read in the second line will eventually set $Win[i]$. Every other process will either terminate before $index = i$, or see $RaceOwner[i]$ set, wait until $Win[i]$ is set, and terminate.

Agreement: In an infinite run, there is a minimum i such that $Win[i]$ is ever set. Every other process must see $RaceOwner[i]$ set, since process i reads $RaceOther[i] = 0$ in line 2 after setting $RaceOwner[i]$ in line 1. So every process will return i.

A fairly straightforward revision implements deadlock-free mutual exclusion. That is, make a copy of these bits for every entrance to the critical section. Each copy has a control bit. The leader is the process who enters the critical section. The exit code is to set the

```
Process i's program
Shared:
    RaceOwner[1..∞], RaceOther[1..∞], Win[1..∞], Lose[1..∞]: arrays of boolean, initially all 0
Local:
    index: integer, initially 1

RaceOwner[i] := 1
if RaceOther[i] = 0 then Win[i] := 1 else Lose[i] := 1 fi
repeat forever
    RaceOther[index] := 1
    if RaceOwner[Index] = 1 then
        await (Win[Index] = 1 or Lose[Index] = 1)
    fi
    if Win[index] = 1 then return(index) fi
    index := index + 1
end repeat
```

Figure 5: (Non-adaptive) leader election for infinite concurrency using bits

control bit. To enter the critical section, a process finds the minimum copy with the control bit still open, contends there until it is the leader or (if it isn't) spins until the control bit is set on that copy, then tries again on the next.

To do starvation-free mutual exclusion, use the index to the current copy of the algorithm to determine who to help next in a (predetermined) diagonalization. ∎

4 Test&set bits

A test&set bit is an object that may take the value 0 or 1, and is initially set to 1. It supports two operations: (1) a reset operation: write 0, and (2) a test&set operation: atomically assign 1 and return the old value. We first make the following observation:

Theorem 4.1 *An infinite number of atomic registers are necessary for implementing a test&set bit when concurrency is bounded, and sufficient when concurrency is infinite.*

Proof: To prove the necessary condition assume to the contrary that such an implementation is possible with a fixed, finite number of atomic registers. This assumption, together with the observation that there exists a trivial deadlock-free mutual exclusion algorithm using a single test&set bit, implies that there is a solution to deadlock-free mutual exclusion with bounded concurrency using a finite number of atomic registers. This contradicts Theorem 3.1. The sufficient condition follows immediately from Theorem 3.4. ∎

Theorem 4.2 *For solving starvation-free mutual exclusion, an infinite number of atomic bits and test&set bits are necessary and sufficient when the concurrency level is bounded.*

Proof: In [Pet94], it is proved that n atomic registers and test&set bits are necessary for solving the starvation-free mutual exclusion problem for n processes (with concurrency level n). This implies the necessary condition above. The sufficient condition follows immediately from Theorem 3.4. ∎

4.1 Naming

The following theorem demonstrates that in certain cases a problem is solvable assuming bounded concurrency, but is not solvable assuming unbounded concurrency. The *wait-free naming* problem is to assign unique names to initially identical processes. After acquiring a name the process may later release it. A process terminates only after releasing the acquired name. A solution to the problem is required to be *wait-free*, that is, it should guarantee that every participating process will always be able to get a unique name and terminate in a finite number of steps regardless of the behavior of other processes (such as abnormal termination).

Theorem 4.3

1. *For bounded concurrency, an infinite number of T&S bits are necessary and sufficient for solving wait-free naming.*
2. *For unbounded concurrency, there is no solution to wait-free naming, even when using an infinite number of T&S bits.*

Proof: Part 1. To prove the sufficient condition, we design the following simple algorithm. We use an infinite number of bits with initial values 0, which are numbered 1, 2, etc. Each process scans the bits, in the same order, starting with the first bit (i.e., bit number 1). At each step, the process applies the `test&set` operation, and either moves to the next bit if the returned value is 1, or stops when the returned value is 0. The process is assigned the name equal to the bit on which its (last) `test&set` operation returned 0. (Notice that wait-freedom is satisfied assuming bounded concurrency, however, it is not satisfied assuming unbounded concurrency since there is a run in which some process will always return 1's and never terminate.)

To prove the necessary condition, we use the following simple argument. Consider an algorithm which solves the problem. Pick a process, run it alone until it acquires a name. Then pick another process and run it alone until it acquires a name, and so on. Since the processes must be assigned unique names, also each state of the algorithm at the time a process terminates must be different from any other such state. Thus, the state space must be infinite and hence also the number of bits.

Part 2. To prove the necessary condition, assume to the contrary that such an algorithm, say A, exists. Let p and p' be two processes. We reach a contradiction by constructing an infinite run α of A in which p and p' always execute the same steps, and hence can never acquire different names. We construct α recursively such that for each positive integer i, α has a prefix $\alpha(i)$ where:

1. p and p' have taken i steps each, and their steps are exactly the same.
2. none of the test&set operations of p and p' has ever returned 0.

We define $\alpha(0)$ to be the empty run. Assume that we have constructed $\alpha(i)$. We show how to extend it to $\alpha(i+1)$. If the next operation of p and p' in $\alpha(i)$, is either a *reset* or a test&set on a bit which is already set to 1, then $\alpha(i+1)$ is constructed by letting p and p' each to take one step, and we are done. Otherwise, their next operation is a test&set operation on a bit, say b, which is set to 0. We will first extend $\alpha(i)$ by letting some other processes take steps until b is set to 1 and only then we let p and p' take one step each, as before.

Setting b to 1 using other processes is the tricky part. We use the following simple observation. If $i+1$ identical processes apply the same operation to the same bit, then the returned values for at least i processes must be the same. Now consider the following extension of $\alpha(i)$. First we activate $i+1$ new processes and let each one of them take one step. Since the processes are initially identical each will apply the same operation to the same bit and hence at least i of them will get the same response. Thus, i processes are still identical. Next we activate these i identical processes and let each one take one step, and are guaranteed to end up with a set of at least $i-1$ identical processes. We repeat this procedure i times, until one process, say q, has taken i steps. We claim that q is identical to both p and p'. This follows from the fact that, as for p and p', process q has taken exactly i steps, and none of the test&set operations of q has ever returned 0.

We now activate q and let it set b to 1, and finally complete the construction of $\alpha(i+1)$ by letting p and p' each take one step. ∎

5 Stronger Objects

5.1 RMW bits

A read-modify-write object supports a read-modify-write operation, in which it is possible to atomically read a value of a shared register and based on the value read, compute some new value and assign it back to the register. When assuming a fault-free model with required participation, many problems become solvable with small constant space.

Theorem 5.1 *There is a solution to consensus with required participation, using 4 RMW bits, with infinite concurrency, even assuming at most one process may fail (by crashing).*

Proof: This is a simple extension to a result in [LA87], which presents an algorithm for n processes, which solves consensus using 4 RMW bits, under the assumption that participation is required and that at most one process may fail (by crashing). We observe that exactly this algorithm works also when assuming an infinite number of processes with infinite concurrency. ∎

We remark that the proof of the theorem above holds only under the assumption that it is known to all processes that there exist processes with identifiers 1 and 2. In the algorithm in [LA87] (which we modify) only one of these processes may be elected.

Theorem 5.2

- *When only a finite number of RMW registers are used for solving starvation-free mutual exclusion with bounded concurrency, one of them must be of unbounded size.*

Process i's program

Shared:

$(ticket, valid)$: integers, initially $(0, 0)$

Local:

$(ticket_i, valid_i)$: integers

$\langle (ticket_i, valid_i) := (ticket, valid)$
$\quad ticket := ticket + 1 \rangle$
while $ticket_i \neq valid_i$ **do**
$\qquad \langle valid_i := valid \rangle$ **od**
critical section
$\langle valid := valid + 1 \rangle$

Figure 6: FIFO mutual exclusion using read-modify-write: The ticket algorithm

- *One unbounded size RMW register is sufficient for solving first-in, first-out adaptive symmetric mutual exclusion when the concurrency level is unbounded.*

Proof: It is shown in [B+82] that in a model which supports a read-modify-write operations any starvation-free mutex for n processes requires at least $\sqrt{2n} + \frac{1}{2}$ values. This result implies the necessary condition.

To prove the sufficient condition, we slightly modify the ticket algorithm from [F+89]. The algorithm in Figure 6 works as follows: A process wishing to enter its critical section takes the next available ticket and waits until its ticket becomes valid, at which point it can safely enter its critical section. When it exits, it discards its ticket and validates the next invalid ticket in order (even if this ticket has not been taken yet).

The shared register contains two fields, *ticket* and *valid*, each a non-negative integer. The brackets $\langle \ \rangle$ are used to explicitly mark the beginning and end of exclusive access to the shared read-modify-write register. An execution of a bracketed section is considered as an atomic action. Each process first reads the value of the shared register, stores its components in local memory, and increments *ticket* by one. At any later point, a process becomes the first in the "waiting line" if it learns (by inspecting *valid*) that its ticket number $ticket_i$ equals *valid*, in which case it can safely enter its critical section. ∎

5.2 Semaphores

Given the results so far about starvation-free mutual exclusion, it is natural to ask whether it can be solved with bounded space? The answer, as presented in [FP87], is that using weak semaphores it can be solved with small constant space for unbounded concurrency, but not with infinite concurrency level.

A binary semaphore is a shared objects that may take the values 0 or 1, and is initially set to 1. It support two operations, called P and V:

- When a process performs a P operation, if the value is 1, then the value is set to 0; otherwise, the process is blocked until the value is 1. Testing and decrementing the semaphore are executed atomically.

- When a process performs a V operation, the value is set to 1.

The result below refers to *weak* semaphores, in which a process that executes a V operation will not be the one to complete the next P operation on that semaphore, if another process has been blocked at that semaphore. Instead, one of the blocked processes is allowed to "pass" the semaphore to a blocked process.

Theorem 5.3 (Friedberg and Peterson [FP87])

- *There is an adaptive symmetric solution to starvation-free mutual exclusion using two atomic bits and two weak semaphores, when the concurrency is unbounded.*
- *There is no solution to the starvation-free mutual exclusion problems using any finite number of atomic registers and weak semaphores, when the concurrency is infinite.*

6 Discussion

We have explored how various assumptions about the number of processes, the concurrency level, and the number of participating processes, effect the design of shared memory algorithms. In particular, we have looked at the case of computing with infinitely many processes. There are many interesting questions that are left open: In the fault-free model using atomic registers only, are there interesting problems that can be solved for any finite and *a priori* known number of processes but cannot be solved for a finite and unknown number of processes (even with unbounded space)? Are there interesting problems that can be solved using a *finite* number of atomic registers by an infinite number of processes with bounded concurrency? Designing wait-free algorithms for problems such as collect, renaming and snapshot assuming unbounded concurrency are challenging problems.

Are there automatic transformations between some algorithms (such as symmetric algorithms) which work correctly for a finite number of processes, to algorithms which work correctly (perhaps assuming bounded concurrency) for infinitely many processes?

Finally, there are algorithms that are correct assuming finite concurrency, but which fail if this constraint is not met? Can we, in general, modify such algorithms to guarantee at least safety (if not liveness) even when the constraints on the number of processes and the concurrency level are not met?

References

[B+82] J. E. Burns, P. Jackson, N. A. Lynch, M. J. Fischer, and G. L. Peterson. Data requirements for implementation of N-process mutual exclusion using a single shared variable. *Journal of the Association for Computing Machinery*, 29(1):183–205, 1982.

[BL93] J. N. Burns and N. A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[CS94] M. Choy and A.K. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[F+89] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Trans. on Programming Languages and Systems*, 11(1):90–114, January 1989.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[FP87] S. A. Friedberg and G. L. Peterson. An efficient solution to the mutual exclusion problem using weak semaphores. *Information Processing Letters*, 25(5):343–347, 1987.

[GK98] E. Gafni and E. Koutsoupias. On uniform protocols. `http://www.cs.ucla.edu/~eli/eli.html`, 1998.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.

[LA87] M. C. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[M+93] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic transactions.* 1993, Morgan Kaufmann.

[Lyn96] N. Lynch. *Distributed algorithms.* 1996, Morgan Kaufmann.

[MT93] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993. (Also published as an AT&T technical memorandum, May 1991.)

[MA95] M. Moir and J. Anderson. Wait-Free algorithms for fast, long-lived renaming, *Science of Computer Programming* 25(1):1–39, 1995.

[Pet94] G. L. Peterson. New bounds on mutual exclusion problems. Technical Report TR68, University of Rochester, February 1980 (Corrected, Nov. 1994).

[SP89] E. Styer and G. L. Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proc. 8th ACM Symp. on Principles of Distributed Computing*, pages 177–191, 1989.

[YA94] J-H. Yang and J.H. Anderson. Time bounds for mutual exclusion and related problems. In *Proc. 26th ACM Symp. on Theory of Computing*, May 1994.