The Black-White Bakery Algorithm
and related bounded-space, adaptive, local-spinning and FIFO algorithms

Gadi Taubenfeld

The Interdisciplinary Center, P.O.Box 167
Herzliya 46150, Israel tgadi@idc.ac.il

Abstract. A mutual exclusion algorithm is presented that has four desired properties: (1) it satisfies FIFO fairness, (2) it satisfies local-spinning, (3) it is adaptive, and (4) it uses finite number of bounded size atomic registers. No previously published algorithm satisfies all these properties. In fact, it is the first algorithm (using only atomic registers) which satisfies both FIFO and local-spinning, and it is the first bounded space algorithm which satisfies both FIFO and adaptivity.
All the algorithms presented are based on Lamport's famous Bakery algorithm [27], which satisfies FIFO, but uses unbounded size registers (and does not satisfy local-spinning and is not adaptive). Using only one additional shared bit, we bound the amount of space required by the Bakery algorithm by *coloring* the tickets taken in the Bakery algorithm. The resulting Black-White Bakery algorithm preserves the simplicity and elegance of the original algorithm, satisfies FIFO and uses finite number of bounded size registers. Then, in a sequence of steps (which preserve simplicity and elegance) we modify the new algorithm so that it is also adaptive to point contention and satisfies local-spinning.

1 Introduction

Motivation and results

Several interesting mutual exclusion algorithms have been published in recent years that are either adaptive to contention or satisfy the local-spinning property [3, 4, 6, 7, 9, 10, 14, 21, 24, 34, 37, 41, 44]. (These two important properties are defined in the sequel.) However, each one of these algorithms either does not satisfy FIFO, uses unbounded size registers, or uses synchronization primitives which are stronger than atomic registers. We presents an algorithm that satisfies all these four desired properties: (1) it satisfies FIFO fairness, (2) it is adaptive, (3) it satisfies local-spinning, and (4) it uses finite number of bounded size atomic registers. The algorithm is based on Lamport's famous Bakery algorithm [27].

The Bakery algorithm is based on the policy that is sometimes used in a bakery. Upon entering the bakery a customer gets a number which is greater than the numbers of other customers that are waiting for service. The holder of the lowest number is the next to be served. The numbers can grow without bound and hence its implementation uses unbounded size registers.

Using only one additional shared bit, we bound the amount of space required in the Bakery algorithm, by *coloring* the tickets taken in the original Bakery algorithm with the colors black and white. The new algorithm, which preserves the simplicity and elegance of the original algorithm, has the following two desired properties, (1) it satisfies FIFO: processes are served in the order they arrive, and (2) it uses finite number of bounded size registers: the numbers taken by waiting processes can grow only up to n, where n is the number of processes.

Then, in a sequence of steps which preserve simplicity and elegance, we modify the new algorithm so that it satisfies two additional important properties. Namely, it satisfies *local-spinning* and is *adaptive* to point contention. The resulting algorithm, which satisfies all theses four properties, is the first algorithm (using only atomic registers) which satisfies both FIFO and local-spinning, and it is the first bounded space algorithm which satisfies both FIFO and adaptivity.

Mutual exclusion

The mutual exclusion problem is to design an algorithm that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65]. It is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the remainder, entry, critical and exit. The problem is to write the code for the entry and the exit sections in such a way that the following two basic requirements are satisfied (assumed a process always leaves its critical section),

Mutual exclusion: *No two processes are in their critical sections at the same time.*

Deadlock-freedom: *If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.*

A stronger liveness requirement than deadlock-freedom is,

Starvation-freedom: *If a process is trying to enter its critical section, then this process must eventually enter its critical section.*

Finally, the strongest fairness requirement is FIFO. In order to formally define it, we assume that the entry section consists of two parts. The first part, which is called the *doorway*, is *wait-free*: its execution requires only bounded number of atomic steps and hence always terminates; the second part is a *waiting* statement: a loop that includes one or more statements. A *waiting process* is a process that has finished the doorway code and reached the waiting part in its entry section.

First-in-first-out (FIFO): *No beginning process can pass an already waiting process. That is, a process that has already passed through its doorway will enter its critical section before any process that has just started.*

Notice that FIFO does not imply deadlock-freedom. (It also does not exactly guarantee bounded bypass, [32] pages 277 and 296.) Throughout the paper, it is assumed that there may be up to n processes potentially contending to enter their critical sections. Each of the n processes has a unique identifier which is a positive integer taken from the set $\{1, ..., n\}$, and the only atomic operations on the shared registers are reads and writes.

Local-spinning

All the mutual exclusion algorithms which use atomic registers (and many algorithms which use stronger primitives) include busy-waiting loops. The idea is that in order to wait, a process *spins* on a flag register, until some other process terminates the spin with a single write operation. Unfortunately, under contention, such spinning may generate lots of traffic on the interconnection network between the process and the memory. Hence, by consuming communication bandwidth spin-waiting by some process can slow other processes.

To address this problem, it makes sense to distinguish between *remote* access and *local* access to shared memory. In particular, this is the case in *distributed shared memory* systems where the shared memory is physically distributed among the processes. I.e., instead of having the "shared memory" in one central location, each process "owns" part of the shared memory and keeps it in its own local memory. For algorithms designed for such systems, it is important to minimize the number of *remote access*. That is, the number of times a process has to reference a shared memory location that does not physically resides on its local memory. In particular, we would like to avoid remote accesses in busy-waiting loops.

Local-spinning: *Local Spinning is the situation where a process is spinning on locally-accessible registers. An algorithm satisfies local-spinning if it is possible to physically distribute the shared memory among the processes in such a way that the only type of spinning required is local-spinning.*

The advantage of local-spinning is that it does not require remote accesses. In the above definition, it does not make any difference if the processes have coherent caches. In cache-coherent machines, a reference to a remote register r causes communication if the current value of r is not in the cache. Since we are interested in proving upper bounds, such a definition would only make our results stronger. (Coherent caching is discussed in Section 4.)

Adaptive algorithms

To speed the entry to the critical section, it is important to design algorithms in which the time complexity is a function of the actual number of contending processes rather than a function of the total number of processes. That is, the time complexity is independent of the total number of processes and is governed only by the current degree of contention.

Adaptive algorithm: *An algorithm is adaptive with respect to time complexity measure ψ, if its time complexity ψ is a function of the actual number of contending processes.*

Our time complexity measures involve counting remote memory accesses. In Section 4, we formally define time complexity w.r.t. two models: one that assumes cache-coherent machines, and another that does not. Our algorithms are also adaptive w.r.t. other common complexity measures, such as *system response time* in which the longest time interval where some process is in its entry section while no process is in its critical section is considered, assuming there is an upper bound of one time unit for step time in the entry or exit sections and no lower bound [38]. In the literature, adaptive, local-spinning algorithms are also called scalable algorithms.

Two notions of contention can be considered: *interval contention* and *point contention*. The interval contention over time interval T is the number of processes that are active in T. The point contention over time interval T is the maximum number of processes that are active at the *same time* in T. Our adaptive algorithms are adaptive w.r.t. both point and interval contention.

Related work

Dijksta's seminal paper [15] contains the first statement and solution of the mutual exclusion problem. Since than it has been extensively studied and numerous algorithms have been published. Lamport's Bakery algorithm is one of the best known mutual exclusion algorithms [27]. Its main appeal lies in the fact that it solves a difficult problem in such a simple and elegant way. All the new algorithms presented in this paper are based on Lamport's Bakery algorithm. For comprehensive surveys of many algorithms for mutual exclusion see [8, 39].

The Bakery algorithm satisfies FIFO, but uses unbounded size registers. Few attempts have been made to bound the space required by the Bakery algorithm. In [43], the integer arithmetic in the original Bakery algorithm is replaced with modulo arithmetic and the *maximum* function and the *less than* relation have been redefined. The resulting published algorithm is incorrect, since it does not satisfy deadlock-freedom. Also in [25], modulo arithmetic is used and the *maximum* function and the *less than* relation have been redefined. In addition, an additional integer register is used. Redefining and explaining these two notions in [25] requires over a full page and involve the details of another unbounded space algorithm. The Black-White Bakery algorithms use integer arithmetic, and do not require to redefine any of the notions used in the original algorithm.

Another attempt to bound the space required by the Bakery algorithm is described in [40]. The algorithm presented is incorrect when the number of processes n is too big; the registers size is bigger than 2^{15} values; and the algorithm is complicated. In [1], a variant of the Bakery algorithm is presents, which uses $3^n + 1$ values per register (our algorithm requires only $2n + 2$ values per register). Unlike the Bakery algorithm (and ours), the algorithm in [1] is not symmetric: process p_i only reads the values of the lower processes. It is possible to replace the unbounded timestamps of the Bakery algorithm (i.e., taking a number) with bounded timestamps, as defined in [22] and constructed in [16, 17, 20], however the resulting algorithm will be rather complex, when the price of implementing bounded timestamps is taken into account.

Several FIFO algorithms which are not based on the Bakery algorithm and use bounded size atomic registers have been published. These algorithms are more complex than the Black-White Bakery algorithm, and non of them is adaptive or satisfies local-spinning. We mention five interesting algorithms below. In [26], an algorithm that requires n (3-valued) shared registers plus two shared bits per process is presented. A modification of the algorithm in [26], is presented in [29] which uses n bits per process. In [30, 31], an algorithm that requires five shared bits per process is presented, which is based on the One-bit algorithm that was devised independently in [12, 13] and [29]. In [42], an algorithm that requires four shared bits per process is presented, which is based on a scheme similar to that of [33]. Finally, in [2] a first-in-first-enabled solution to the ℓ-exclusion problem is presented using bounded timestamps. We are not aware of a way to modify these algorithms, so that they satisfy adaptivity and local-spinning.

In addition to [27], the design of the Black-White Bakery algorithm was inspired by two other papers [18, 19]. In [18], an ℓ-exclusion algorithm for the FIFO allocation of ℓ identical resources is presented, which uses a single read-modify-write object. The algorithm uses colored tickets where the number of different colors used is only $\ell+1$, and hence only *two* colors are needed for mutual exclusion. In [19], a starvation-free solution to the mutual exclusion problem that uses *two* weak semaphores (and two shared bits) is presented.

Three important papers which have investigated local-spinning are [9, 21, 34]. The various algorithms presented in these papers use strong synchronization primitives (i.e., stronger than atomic registers), and require only a constant number of remote accesses for each access to a critical section. Performance studies done in these papers have shown that local-spinning algorithms scale well as contention increases. More recent local-spinning algorithms using objects which are stronger than atomic registers are presented in [24, 41], these algorithms have unbounded space complexity. Local-spinning algorithms using only atomic registers are presented in [4–6, 44], and a local-spinning algorithm using only non-atomic registers is presented in [7], these algorithms do not satisfy FIFO.

The question whether there exists an adaptive mutual exclusion algorithm using atomic registers was first raised in [36]. In [35], it is shown that is no such algorithm when time is measured by counting all accesses to shared registers. In [10, 14, 37] adaptive algorithms using atomic registers, which do not satisfy local-spinning, are presented. In [4, 6], local-spinning and adaptive algorithms are presented. None of these adaptive algorithms satisfy FIFO. In [3], an interesting technique for collecting information is introduced, which enables to transform the Bakery algorithm [27] into its corresponding adaptive version. The resulting FIFO algorithm is adaptive, uses unbounded size registers and does not satisfy local-spinning. We use this technique to make our algorithms adaptive.

The time complexity of few known adaptive and/or local-spinning non-FIFO algorithms, and in particular the time complexity of [6], is better than the time complexity of our adaptive algorithms. This seems to be the prices to be paid for satisfying the FIFO property. We discuss this issue in details in Section 5.

2 Lamport's Bakery Algorithm

We first review Lamport's Bakery algorithm [27]. The algorithm uses a boolean array *choosing*$[1..n]$, and an integer array *number*$[1..n]$ of *unbounded* size registers. The entries $choosing_i$ and $number_i$ can be read by all the processes but can be written only by process i. The relation "$<$" used in the algorithm on ordered pairs of integers is the *lexicographic order* relation and is defined by $[a, b] < [c, d]$ if $a < c$, or if $a = c$ and $b < d$. The statement **await** *condition* is used as an abbreviation for **while** $\neg$*condition* **do** *skip*. The algorithm is given below.

Algorithm 1. The Bakery Algorithm: `process` i`'s code`

```
Shared variables:
    $choosing[1..n]$: boolean array
    $number[1..n]$: array of type $\{0, ..., \infty\}$
    Initially $\forall i : 1 \le i \le n : choosing_i =$ false and $number_i = 0$

    $choosing_i :=$ true                                  /* beginning of doorway */
    $number_i := 1 + \text{maximum}(\{number_j \mid 1 \le j \le n\})$
    $choosing_i :=$ false                                       /* end of doorway */
    for $j = 1$ to $n$ do
        await $choosing_j =$ false
        await $(number_j = 0) \lor ([number_j, j] \ge [number_i, i])$
    od
    critical section
    $number_i := 0$                                                 /* exit code */
```

As Lamport has pointed out, the correctness of the Bakery algorithm depends on how the maximum is computed [28]. We assume a simple correct implementation in which a process first reads into local memory all the n *number* registers, one at a time, and then computes the maximum over these n values.

3 The Black-White Bakery Algorithm

Using only one additional shared bit, called *color* of type $\{$`black`, `white`$\}$, we bound the amount of space required in the Bakery algorithm, by *coloring* the tickets taken with the colors black and white. In the new algorithm, the numbers of the tickets used can grow only up to n, where n is the number of processes.

The first thing that process i does in its entry section is to take a colored ticket $ticket_i = (mycolor_i, number_i)$, as follows: i first reads the shared bit *color*, and sets its ticket's color to the value read. Then, it takes a number which is greater than the numbers of the tickets which have the same color as the color of its own ticket. Once i has a ticket, it waits until its colored ticket is the *lowest* and then it enters its critical section. The order between colored tickets is defined as follows: If two tickets have different colors, the ticket whose color is *different* from the value of the shared bit *color* is smaller. If two tickets have the same color, the ticket with the smaller number is smaller. If tickets of two processes have the same color and the same number then the process with the smaller identifier enters its critical section first. Next, we explain when the shared *color* bit is written. The first thing that a process i does when it leaves its critical section (i.e., its first step in the exit section) is to set the *color* bit to a value which is different from the color of its ticket. This way, i gives priority to waiting processes that hold tickets with the same color as the color of i's ticket.

Until the value of the *color* bit is first changed, all the tickets have the same color, say white. The first process to enter its critical section flips the value of the *color* bit (i.e., changes it to black), and hence the color of all the new tickets taken thereafter (until the color bit is modified again) is black. Next, *all* the processes which hold white colored tickets enter and then exit their critical sections one at a time until there are no processes holding white tickets in the system. Only then the process with the lowest black ticket is allowed to enter its critical section, and when it exits it changes to white the value of the *color* bit, which gives priority to the processes with black tickets, and so on.

Three data structures are used: (1) a single shared bit named *color*, (2) a boolean array *choosing*$[1..n]$, and (3) an array with n entries where each entry is a colored ticket which ranges over $\{\texttt{black}, \texttt{white}\} \times \{0, ..., n\}$. We use $mycolor_i$ and $number_i$ to designate the first and second components, respectively, of the ordered pair stored in the i^{th} entry.

Algorithm 2. The Black-White Bakery Algorithm: `process` i`'s code`

Shared variables:

```
    color: a bit of type {black, white}
    choosing[1..n]: boolean array
    (mycolor, number)[1..n]: array of type {black, white} × {0, ..., n}
    Initially ∀i : 1 ≤ i ≤ n : choosing_i = false and number_i = 0,
    the initial values of all the other variables are immaterial.

1   $choosing_i := \texttt{true}$                                  /* beginning of doorway */
2   $mycolor_i := color$
3   $number_i := 1 + \max(\{number_j \mid (1 \leq j \leq n) \wedge (mycolor_j = mycolor_i)\})$
4   $choosing_i := \texttt{false}$                                 /* end of doorway */
5   for $j = 1$ to $n$ do
6       await $choosing_j = \texttt{false}$
7       if $mycolor_j = mycolor_i$
8       then await $(number_j = 0) \vee ([number_j, j] \geq [number_i, i]) \vee$
                   $(mycolor_j \neq mycolor_i)$
9       else await $(number_j = 0) \vee (mycolor_i \neq color) \vee$
                   $(mycolor_j = mycolor_i)$ fi
10  od
11  critical section
12  if $mycolor_i = \texttt{black}$ then $color := \texttt{white}$ else $color := \texttt{black}$ fi
13  $number_i := 0$
```

In line 1, process i indicates that it is contending for the critical section by setting its *choosing* bit to true. Then it takes a colored ticket by first "taking" a color (step 2) and then taking a number which is greater by one than the numbers of the tickets with the same color as its own (step 3). For computing the maximum, we assume a simple implementation in which a process first reads into local memory all the n tickets, one at a time atomically, and then computes the maximum over numbers of the tickets with the same color as its own.

After passing the doorway, process i waits in the *for loop* (lines 5–10), until it has the lowest colored ticket and then it enters its critical section. We notice that each one of the three terms in each of the two await statements is evaluated separately. In case processes i and j have tickets of the same color (line 8), i waits until it notices that either (1) j is not competing any more, (2) i has a smaller number, or (3) j has reentered its entry section. (If two processes have the same number then the process with the smaller identifier enters first.) In case processes i and j have tickets with different colors (line 9), i waits until it notices that either (1) j is not competing any more, (2) i has priority over j because i's color is *different* than the value of the color bit, or (3) j has reentered its entry section.

In the exit code (line 12), i sets the *color* bit to a value which is different than the color of its ticket, and sets its ticket number to 0 (line 13). The algorithm is also correct if we replace the order of lines 11 and 12, allowing process i to write the color bit immediately before it enters its critical section. We observe that the order of lines 12 and 13 is crucial for correctness; and that without the third clause in the await statement in line 9 the algorithm can deadlock. Although the color bit is not a purely single-writer registers, there is at most one write operation pending on it at any time.

The following lemma captures the effect of the tickets' colors on the order in which processes enter their critical sections. For lack of space all the proofs are omitted from this abstract.

Lemma 1. *Assume that at time t, the value of the color bit is $\mathtt{c} \in \{black, white\}$. Then, any process which at time t is in its entry section and holds a ticket with a color different than $\mathtt{c}$ must enter its critical section before any process with a ticket of color $\mathtt{c}$ can enter its critical section.*

For example, if the value of the *color* bit is white, then no process with a white ticket can enter its critical section until all the processes which hold black tickets enter their critical sections. The following corollary follows immediately from Lemma 1.

Corollary 1. *Assume that at time t, the value of the color bit has changed from $\mathtt{c} \in \{black, white\}$ to the other value. Then, at time t, every process that is in its entry section has a ticket of color $\mathtt{c}$.*

The following theorem states the main properties of the algorithm.

Theorem 1. *The Black-White Bakery Algorithm satisfies mutual exclusion, deadlock-freedom, FIFO, and uses finite number of bounded size registers (each of size one bit or* $\log(2n+2)$ *bits).*

4 Adaptive FIFO Algorithm with Bounded Space

In [3], a new object, called an *active set* was introduced, together with an implementation which is wait-free, adaptive and uses only bounded number of bounded size atomic registers. Notice that wait-freedom implies local spinning, as a wait-free implementation must also be spinning-free. The authors of [3], have shown how to transform the Bakery algorithm into its corresponding adaptive version using the active set object. We use the same efficient transformation.

Active set: *An active set S object supports the following operations:*

- `join(S)`*: which adds the id of the executing process to the set S. That is, when process i executes this operation the effect is to execute, $S := S \cup \{i\}$.*
- `leave(S)`*: which removes the id of the executing process from the set S. That is, when process i executes this operation the effect is to execute, $S := S - \{i\}$.*
- `getset(S)`*: which returns the current set of active processes. More formally, the following two conditions must be satisfied,*
  - *the set returned includes all the processes that have finished their last* `join(S)` *before the current* `getset(S)` *has started, and did not start* `leave(S)` *in the time interval between their last* `join(S)` *and the end of the current* `getset(S)`*.*
  - *the set returned does not includes all the processes that have finished their last* `leave(S)` *before the current* `getset(S)` *has started, and did not start* `join(S)` *in the time interval between their last* `leave(S)` *and the end of the current* `getset(S)`*.*

The implementation in [3] of the active set object is both wait-free and adaptive w.r.t. the number of steps required. That is, the number of steps depends only on the number of active processes – the number of processes that finished `join(S)` and have not yet started `leave(S)`. Next we transform the Black-white Bakery algorithm into its corresponding adaptive version. The basic idea is to use an active set object in order to identify the active processes and then to ignore the other processes. The code of the *adaptive* Black-White Bakery algorithm (Algorithm 3) is shown on the next page.

For computing the maximum, we assume that a process first reads into local memory *only* the tickets of processes in S, one at a time atomically, and then computes the maximum over numbers of the tickets with the same color as its own. Algorithm 3 is adaptive only if we assume that spinning on a variable while its value does not change, is counted only as one operation (i.e., only remote uncached accesses are counted.) In the next section we modify the algorithm so that it is adaptive even without the above assumption.

In order to be able to formally claim that Algorithm 3 is adaptive, we need to formally define time complexity. As discussed in the introduction, for certain shared memory systems, it makes sense to distinguish between *remote* and *local* access to shared memory. Shared registers may be locally-accessible as a result of coherent caching, or when using distributed shared memory where shared memory is physically distributed among the processors.

Algorithm 3. THE ADAPTIVE BLACK-WHITE BAKERY ALGORITHM: i's `code`

Shared variables:

```
    $\boxed{S\text{: adaptive active set, initially } S = \emptyset}$
    $color$: a bit of type $\{\texttt{black}, \texttt{white}\}$
    $choosing[1..n]$: boolean array
    $(mycolor, number)[1..n]$: array of type $\{\texttt{black}, \texttt{white}\} \times \{0, ..., n\}$
    Initially $\forall i : 1 \leq i \leq n : choosing_i = \texttt{false}$ and $number_i = 0$,
    the initial values of all the other variables are immaterial.
                                                   /* beginning of doorway */
1   $\boxed{join(S)}$                                            /* $S := S \cup \{i\}$ */
2   $choosing_i := \texttt{true}$
3   $\boxed{localS := getset(S) - \{i\}}$           /* reads S into local variable */
4   $mycolor_i := color$
5   $number_i := 1 + \max(\{number_j \mid \boxed{(j \in localS)} \wedge (mycolor_j = mycolor_i)\})$
6   $choosing_i := \texttt{false}$
7   $\boxed{localS := getset(S) - \{i\}}$           /* reads S into local variable */
                                                         /* end of doorway */
8   $\boxed{\textbf{for every } j \in localS \textbf{ do}}$
9       await $choosing_j = \texttt{false}$
10      if $mycolor_j = mycolor_i$
11      then await $(number_j = 0) \vee ([number_j, j] \geq [number_i, i]) \vee$
                   $(mycolor_j \neq mycolor_i)$
12      else await $(number_j = 0) \vee (mycolor_i \neq color) \vee$
                   $(mycolor_j = mycolor_i)$ fi
13  od
14  critical section
15  if $mycolor_i = \texttt{black}$ then $color := \texttt{white}$ else $color := \texttt{black}$ fi
16  $number_i := 0$
17  $\boxed{leave(S)}$                                           /* $S := S - \{i\}$ */
```

Remote access: *We define a remote access by process p as an attempt to access a memory location that does not physically resides on p's local memory. The remote memory location can either reside in a central shared memory or in some other process' memory.*

Next, we define when remote access causes *communication.*

Communication: *Two models are possible,*

1. Distributed Shared Memory (DSM) Model: *Any remote access causes communication;*
2. Coherent Caching (CC) Model: *A remote access to register r causes communication if (the value of) r is not (the same as the value) in the cache. That*

is, communication is caused only by a remote write access that overwrites a different process' value or by the first remote read access by a process that detects a value written by a different process.

It is important to notice that spinning on a remote variable while its value does not change, is counted only as one remote operation that causes communication in the CC model, while it is counted as many operations that causes communication in the DSM model. Next we define time complexity. This complexity measure is defined with respect to either the DSM Model or the CC model, and whenever it is used, we will say explicitly which model is assumed.

Time complexity: *The maximum number of remote accesses which cause communication that a process, say p, may need to perform in its entry and exit sections in order to enter and exit its critical section since p started executing the code of its entry section.*

Theorem 2. *Algorithm 3 satisfies mutual exclusion, deadlock-freedom, FIFO, uses finite number of bounded size registers, and is adaptive w.r.t. time complexity in the CC model.*

Algorithm 3 is adaptive in the CC model, even if it is assumed that every write access causes communication. The Bakery algorithm uses single-writer safe registers. Our adaptive algorithm requires using multi-writer registers and atomic registers. The following results show that this is unavoidable.

Theorem 3 (Anderson and Kim [7]). *There is no adaptive mutual exclusion algorithms, in both the CC and the DSM models, if registers accesses are non-atomic.*

Theorem 4. *There is no adaptive mutual exclusion algorithm, in both the CC and the DSM models, using only single-writer registers.*

Algorithm 3 is not adaptive w.r.t. time complexity in the DSM model, and it does not satisfy local-spinning. This is due to the fact that in Algorithm 3 two processes may spin on the same shared variable. Our next algorithm satisfies these two additional properties: (1) it is adaptive also w.r.t. time complexity in the DSM model, and (2) it satisfies local-spinning.

5 Adaptive and Local-spinning Black-White Bakery Alg.

We modify Algorithm 3, so that the new algorithm is: (1) adaptive w.r.t. time complexity in the DSM model, (2) satisfies local-spinning, (3) satisfies FIFO, and (4) uses bounded space. In Algorithm 3, process i may need to busy-wait for another process, say j, in one of two cases:

1. Process i might need to wait until the value of $choosing_j$ changes (line 9).
2. Process i has lower priority than j and hence i has to wait until j exits its critical section.

Algorithm 3 does not satisfy local-spinning since in each one of these two cases process i waits by spinning on remote registers. To overcome this difficulty, in Algorithm 4, process i uses two new single-reader shared bits, $spin.ch[i,j]$ and $spin.nu[i,j]$, which are both assumed to be locally accessible for process i.

1. In the first case, instead of spinning on $choosing_j$, process i spins locally on $spin.ch[i,j]$, waiting for j to notify it that the value of $choosing_j$ has been changed. Process j notifies i of such a change by writing into $spin.ch[i,j]$.
2. In the second case, instead of waiting for j to exit its critical section by spinning on the variables $number_j$, $color$ and $mycolor_j$, process i spins locally on $spin.nu[i,j]$, waiting for j to notify it that j has exited its critical section. Process j notifies i when it exits by writing into $spin.nu[i,j]$.

To implement all the (single-reader) spin bits, we use the two dimensional arrays *spin.ch* and *spin.nu*. To keep the algorithm adaptive we use one active set S which records at any moment the set of active processes. As in Algorithm 3, a process uses S in order to know which processes are concurrent with it when it either takes a number or when it compares its ticket with the tickets of the other active processes. In addition, in Algorithm 4, the adaptive active set S is used to know which are the waiting processes that need to be notified of a change in one of the shared variables. The code of the *adaptive and local-spinning* Black-White Bakery algorithm (Algorithm 4) is shown on the next page.

Theorem 5. *Algorithm 4 satisfies mutual exclusion, deadlock-freedom, FIFO, uses finite number of bounded size registers, is adaptive w.r.t. time complexity in the DSM model, and satisfies local-spinning.*

The time complexity in the CC model of both Algorithms 3 and Algorithm 4, is dominated by the complexity of the active set, and is $O(\max(k, comp.S))$, where k is the point contention and $comp.S$ is the step complexity of the active set. Since Algorithm 3 does not satisfy local-spinning its time complexity in the DSM model is unbounded, however, the time complexity of Algorithm 4 is $O(\max(k, comp.S))$ also in the DSM model. The step complexity of the active set implementation from [3] is $O(k^4)$. However, a more efficient implementation exists which has only $O(k^2)$ step complexity [11, 23]. (This is an implementation of *collect* which is a stronger version of active set.) Thus, using this implementation, the time complexity of Algorithm 4 is $O(k^2)$ for both the CC and DSM model, where k is the point contention. As already mentioned, few other adaptive algorithms which do not satisfy FIFO have better time complexity.

The time complexity of the algorithm in [6] is $O(\min(k, \log n))$ for both the CC and DSM model, where k is point contention (this is also its system response time). The time complexity of the algorithm in [4] is $O(\min(k^2, k \log n))$ for both the CC and DSM model, however here k is interval contention. The time complexity of the algorithm in [3] is $O(k^4)$ for the CC mode, and since it does not satisfy local-spinning its time complexity in the DSM model is unbounded. The time complexity of the algorithm in [14] for the CC model is $O(N)$, however its system response time is $O(k)$. In [10], it is assumed that busy-waiting is counted as a single operation (even if the value of the lock changes several times while waiting). The step complexity of the algorithm in [10] is $O(k)$ and its system response time is $O(\log k)$. The system response time of the algorithm in [37] (which works for infinitely many processes) is $O(k)$.

Algorithm 4. THE ADAPTIVE AND LOCAL-SPINNING BLACK-WHITE BAKERY ALGORITHM: `process` i`'s code`

Shared variables:

S: adaptive active set, initially $S = \emptyset$
spin.ch$[1..n, 1..n]$: two dimensional boolean array `/*spin on choosing*/`
spin.nu$[1..n, 1..n]$: two dimensional boolean array `/* spin on number */`
color: a bit of type $\{$`black`$,$ `white`$\}$
choosing$[1..n]$: boolean array
$(mycolor, number)[1..n]$: array of type $\{$`black`$,$ `white`$\} \times \{0, ..., n\}$
Initially $\forall i : 1 \leq i \leq n : choosing_i =$ `false` and $number_i = 0$,
the initial values of all the other variables are immaterial.

```
                                                  /* beginning of doorway */
1   $join(S)$                                                /* $S := S \cup \{i\}$ */
2   $choosing_i :=$ true
3   $localS := getset(S) - \{i\}$              /* reads S into local variable */
4   $mycolor_i := color$
5   $number_i := 1 + \max(\{number_j \mid (j \in localS) \wedge (mycolor_j = mycolor_i)\})$
6   $choosing_i :=$ false
7   $localS := getset(S) - \{i\}$ /* notifyAll that $choosing_i$ has changed */
8   for every $j \in localS$ do $spin.ch[j, i] :=$ false od
                                                        /* end of doorway */
9   for every $j \in localS$ do
10      $spin.ch[i, j] :=$ true          /* waits until $choosing_i =$ false */
11      if $choosing_j =$ true then await $spin.ch[i, j] =$ false fi
12      $spin.nu[i, j] :=$ true    /* writes first to avoid race cond. */
13      if $mycolor_j = mycolor_i$ /* waits until $i$ has priority over $j$ */
14      then if $(number_j = 0) \vee ([number_j, j] \geq [number_i, i]) \vee$
                    $(mycolor_j \neq mycolor_i)$
15              then skip else await $spin.nu[i, j] =$ false fi
16      else  if $(number_j = 0) \vee (mycolor_i \neq color) \vee (mycolor_j = mycolor_i)$
17              then skip else await $spin.nu[i, j] =$ false fi
18      fi
19  od
20  critical section
21  if $mycolor_i =$ black then $color :=$ white else $color :=$ black fi
22  $number_i := 0$
23  $leave(S)$                                                /* $S := S - \{i\}$ */
24  $localS := getset(S)$                          /* notifyAll of $i$'s exit */
25  for every $j \in localS$ do $spin.nu[j, i] :=$ false od
```

References

1. U. Abraham. Bakery algorithms. In *Proc. of the Concurrency, Specification and Programming Workshop*, pages 7–40, 1993.
2. Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Transactions on Programming Languages and Systems*, 16(3):939–953, 1994.
3. Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In *Proc. 40th IEEE Symp. on Foundations of Computer Science*, 262–272, 1999.
4. Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 30:67–86, 2002.
5. J. H. Anderson. A fine-grained solution to the mutual exclusion problem. *Acta Informatica*, 30(3):249–265, 1993.
6. J.H. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. *Proceedings of the 14th international symposium on distributed computing. Lecture Notes in Computer Science*, 1914:29–43, oct 2000.
7. J.H. Anderson and Y.-J. Kim. Nonatomic mutual exclusion with local spinning. In *Proc. 21st ACM Symp. on Principles of Distributed Computing*, pages 3–12, 2002.
8. J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:75–110, 2003.
9. T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessor. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):6–16, 1990.
10. H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. *Distributed Computing*, 15(3):177–189, 2002.
11. H. Attiya and A. Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):144–468, 2003.
12. J. E. Burns and A. N. Lynch. Mutual exclusion using indivisible reads and writes. In *18th annual allerton conf. on comm., control and computing*, 833–842, 1980.
13. J. N. Burns and N. A. Lynch. Bounds on shared-memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.
14. M. Choy and A.K. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.
15. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
16. D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, 1997.
17. C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible! In *Proc. 24rd ACM Symp. on Theory of Computing*, pages 655–666, May 1992.
18. 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.
19. S. A. Friedberg and G. L. Peterson. An efficient solution to the mutual exclusion problem using weak semaphores. *Info. Processing Letters*, 25(5):343–347, 1987.
20. R. Gawlick, N. A. Lynch, and N. Shavit. Concurrent timestamping made simple. In *Israel Symposium on Theory of Computing Systems*, pages 171–183, 1992.
21. G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computers*, 28(6):69–69, June 1990.
22. A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, 1993.

23. M. Inoue, S. Umetani, T. Masuzawa, and H. Fujiwara. Adaptive long-lived $O(k^2)$-renaming with $O(k^2)$ steps. In *15th international symposium on distributed computing*, 2001. *LNCS 2180* Springer Verlag 2001, 123–135.
24. P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proc. 22nd ACM Symp. on Principles of Distributed Computing*, pages 295–304, July 2003.
25. P. Jayanti, K. Tan, G. Friedland, and A. Katz. Bounding Lamport's Bakery algorithm. In *28 annual conference on current trends in theory and practice of informatics*, December 2001. *LNCS 2234* Springer Verlag 2001, 261–270.
26. H.P. Katseff. A new solution to the critical section problem. In *Proc. 10th ACM Symp. on Theory of Computing*, pages 86–88, May 1978.
27. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.
28. L. Lamport. A bug in the Bakery algorithm. Technical Report CA–7704–0611, Massachusette computer associates, inc., April 1977.
29. L. Lamport. The mutual exclusion problem: Part II – statement and solutions. *Journal of the ACM*, 33:327–348, 1986.
30. E. A. Lycklama. A first-come-first-served solution to the critical section problem using five bits. M.Sc. thesis, University of Toronto, October 1987.
31. E. A. Lycklama and V. Hadzilacos. A first-come-first-served mutual exclusion algorithm with small communication variables. *ACM Trans. on Programming Languages and Systems*, 13(4):558–576, 1991.
32. N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., 1996.
33. J. M. Morris. A starvation-free solution to the mutual exclusion problem. *Information Processing Letters*, 8(2):76–80, 1979.
34. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, 1991.
35. R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, December 1992.
36. M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993. (Published as an AT&T technical memorandum, May 1991.)
37. M. Merritt and G. Taubenfeld. Computing with infinitely many processes. In *14th international symposium on distributed computing*, October 2000. *LNCS 1914* Springer Verlag 2000, 164–178.
38. G. L. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proc. 9th ACM Symp. on Theory of Computing*, pages 91–97, 1977.
39. M. Raynal. *Algorithms for mutual exclusion.* The MIT Press, 1986.
40. S. Vijayaraghavan. A variant of the bakery algorithm with bounded values as a solution to Abraham's concurrent programming problem. In *Proc. of Design, Analysis and Simulation of Distributed Systems*, 2003.
41. M.L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proc. 21th ACM Symp. on Principles of Distributed Computing*, pages 31–40, July 2002.
42. B. K. Szymanski. Mutual exclusion revisited. In *Proc. of the 5th Jerusalem Conf. on Information Technology*, pages 110–117, October 1990.
43. T. Woo. A note on Lamport's mutual exclusion algorithm. *Operating Systems Review (ACM)*, 24(4):78–80, October 1990.
44. J-H. Yang and J.H. Anderson. Fast, scalable synchronization with minimal hardware support. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pages 171–182, 1993.