Technion - Israel Institute of Technology
Computer Science Department

Adaptive Long-Lived Renaming with Read and Write Operations

by

Hagit Attiya and Arie Fouren

Technical Report #0956
March 1999

Adaptive Long-Lived Renaming with Read and Write Operations*

(Preliminary Version)

Hagit Attiya and Arie Fouren
Department of Computer Science
The Technion
Haifa 32000, Israel

March 1, 1999

Abstract

This paper presents an adaptive algorithm for long-lived renaming using only read and write operations. A process p_i obtains a new name in the range $\{1, \ldots, k(2k-1)\}$, where k is the maximal number of processes simultaneously participating with p_i; the number of steps performed by p_i is $O(k^2 \log k)$. The range of names is reduced to $\{1, \ldots, 6k-1\}$, by employing the algorithm of Burns and Peterson.

1 Introduction

Distributed coordination algorithms are designed to accommodate a large number of processes, each with a distinct identifier; in order to coordinate, processes must obtain up-to-date information from each other. A *wait-free* algorithm guarantee that a process completes its operation within a finite number of its own steps regardless of the behavior of other processes; in a typical wait-free algorithm, information is collected by reading from an array indexed with process' identifiers. This scheme is an over-kill in a well-designed system where often only a few processes simultaneously participate in the coordination algorithm [12]: Many entries are read from the array although they contain irrelevant information about processes not wishing to coordinate.

An attractive approach to this problem is to *rename* the participating processes [2, 13]. A (long-lived) renaming algorithm provides two operations, getName and releaseName. Before starting the coordination, a process uses getName to obtain a unique *new name*—a positive

*Contact author: Hagit Attiya, email: hagit@cs.technion.ac.il, phone: +972-4-8294365.

integer, bounded by a function of the current number of processes. The process then performs the coordination algorithm, using the new name instead of its identifier. When the coordination algorithm completes, releaseName allows the name to be re-claimed later by other processes.

A renaming algorithm guarantees *adaptive name space* if the names are bounded by a function of the current number of participating processes. For the renaming stage to be useful it must also have *adaptive step complexity*: The number of steps a process takes in order to obtain a new name (or to release it) is bounded by a function of the maximal number of processes *simultaneously* trying to obtain new names. Under this definition, a getName operation is delayed only when a large group of processes participate simultaneously. For example, the step complexity of the getName operation *op* depicted in Figure 1 should be constant since at most two processes simultaneously participate at each point during the operation. This requirement is made although a large number of processes acquire and release names throughout *op*, and many processes participate simultaneously just before *op* starts.

Several renaming algorithms are known [2, 4, 6, 8, 13, 14] and some of them provide an adaptive name space. However, none of these algorithms has adaptive step complexity: The algorithm of Burns and Peterson, for example, requires processes to read from an array indexed with process' identifiers.

There are algorithms which enjoy adaptive step complexity: Choy and Singh [9] present mutual exclusion algorithms whose step complexity is adaptive when amortized across many entries to the critical section; however, these algorithms are not wait-free. Afek, Dauber and Touitou [1] show wait-free object implementations with adaptive step complexity; however, they employ load-linked and store-conditional operations, which are not available in many distributed environments and cannot be implemented from more basic read and write operations [11]. Attiya and Fouren [3, 10] give wait-free algorithms for renaming and collecting information with adaptive step complexity, using only reads and writes; however, the step complexity of these algorithms depends on the total number of operations performed so far, and their complexity does not decrease when the number of participating processes drops. Moreover, the renaming algorithms of [3] can be invoked only once by each process.

This paper presents adaptive algorithms for long-lived renaming, using read and write operations. The step complexity of getName and releaseName dynamically adjusts to contention; that is, it is bounded by a function of the maximal number of processes *simultaneously* exe-

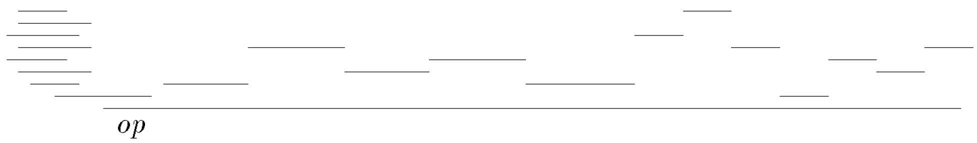


Figure 1: Overlapping getName operations; each line represents an operation; different processes appear at different horizontal levels.

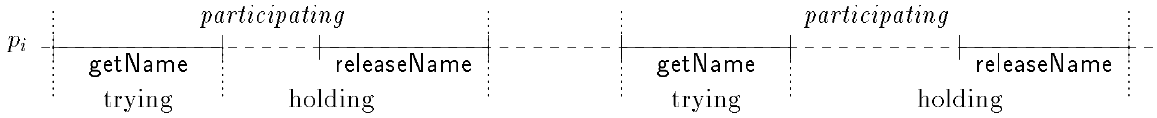


Figure 2: Invocations of getName and releaseName.

cuting the algorithm at some point during the getName operation. We present an algorithm in which a process p_i obtains a name in the range $\{1, \ldots, k(2k-1)\}$, where k is the maximal contention during the getName operation of p_i; the number of steps performed by p_i is $O(k^2 \log k)$. The algorithm of Burns and Peterson [8] is employed in order to reduce the range to be $\{1, \ldots, 6k-1\}$. (This algorithm appears in Appendix 5.) Both algorithms require an unbounded amount of shared memory.

Our renaming algorithms are truly adaptive; for example, the step complexity of the operation depicted in Figure 1 is constant. Unfortunately, we were not able to obtain truly adaptive algorithms for other problems, most notably, for collecting up-to-date information. Additional ideas seem to be needed in order to obtain such algorithms or to prove their inexistence.

2 Preliminaries

We assume a standard asynchronous shared-memory model of computation, following, e.g., [11]. A system consists of n processes, $p_1, \ldots, p_n$, communicating by reading and writing to shared registers; we assume that each process can read from and write to any register (*multi-writer multi-reader* registers), and that there is an infinite number of registers.

In the *long-lived M-renaming* problem, processes repeatedly acquire and release distinct names; the names must be in the range $\{1, \ldots, M\}$. In more detail, a solution to the problem must supply two procedures, $\mathsf{getName}_i$ and $\mathsf{releaseName}_i$, for each process p_i. Procedure $\mathsf{getName}_i$ returns a *new name*, y. A process alternates between invoking $\mathsf{getName}_i$ and $\mathsf{releaseName}_i$, starting with $\mathsf{getName}_i$.

Consider some execution of a long-lived renaming algorithm α; below, α' is some finite prefix of α.

Process p_i is *participating* at the end of α', if α' includes an invocation of $\mathsf{getName}_i$ without a return from the matching $\mathsf{releaseName}_i$. Process p_i, participating at the end of α', can either be *trying* to obtain a new name, that is, p_i has not returned from $\mathsf{getName}_i$, or already *holding* a name, that is, p_i has already returned from $\mathsf{getName}_i$. In the latter case, we say that p_i is *named y* at the end of α' if the last invocation of $\mathsf{getName}_i$ returned y. (See Figure 2.)

The main requirement from a long-lived renaming algorithm is the following:

Uniqueness: If participating process p_i is named y_i and participating process p_j ($j \neq i$) is named y_j at the end of α', then $y_i \neq y_j$.

We wish to find an M-renaming algorithm with a range of names, M, and step complexity that depend only on the contention during the execution of getName.

The *active processes* at the end of α', denoted $\mathrm{Cont}(\alpha')$, is the set of processes participating at the end of α'. The *contention* at the end of α' is $|\mathrm{Cont}(\alpha')|$, denoted $\mathrm{Cont\#}(\alpha')$.

Consider a finite execution interval β of α; we can write $\alpha = \alpha_1\beta\alpha_2$. The contention during β, denoted $\mathrm{Cont\#}(\beta)$, is the maximum contention over all prefixes $\alpha_1\beta'$ of $\alpha_1\beta$. If the contention during β is k, then for some prefix β' of β, $\mathrm{Cont\#}(\alpha_1\beta') = k$.

A renaming algorithm has an *adaptive name space* if there is a function M, such that the name obtained in an execution interval of getName, β, is in the range $\{1, \ldots, \mathrm{M}(\mathrm{Cont\#}(\beta))\}$.

A renaming algorithm has *adaptive step complexity* if there is a bounded function S, such that the number of steps performed by p_i in any execution interval of $\mathsf{getName}_i$, β, and in the matching $\mathsf{releaseName}_i$ is at most $\mathrm{S}(\mathrm{Cont\#}(\beta))$.

The contention during an execution interval is clearly bounded by n, the total number of processes. Therefore, in an adaptive algorithm, $\mathsf{getName}_i$ and $\mathsf{releaseName}_i$ terminate within a bounded number of steps of p_i, regardless of the behavior of other processes. Thus, an algorithm with adaptive step complexity is necessarily *wait-free*.

3 The Renaming Algorithm

The algorithm uses a sequence of *sieves*, numbered $1, 2, \ldots$. A process tries to get a name in the sieves, one after the other, until successful in some sieve. The name obtained from sieve s is a pair containing the sieve number, s, and the rank of the process among the processes possibly getting a name from the sieve. As we prove below, a process p_i accesses sieve s only if there are $O(s)$ distinct processes which are simultaneously participating at some point during p_i's getName operation. Thus, each sieve "catches" at least one of the concurrently participating processes.

The crux of the algorithm is the way sieves are implemented. This uses ideas from the Borowsky-Gafni simulation [5, 7], modified to guarantee adaptive step complexity.

Each sieve has an infinite number of copies. However, at each point in the execution, participating processes are only "inside" a single copy of a sieve; these processes have overlapping executions of getName. A process accessing sieve s tries to get inside a sieve by checking if it is among the first processes to access the current copy of this sieve. The process succeeds only if this copy is still free (no other process is already inside it), and all processes which could get a name from the previous copy of the sieve have already left it.

If a process cannot get inside a sieve, then some concurrent process either gets a name whose first component is s (either a process already with a name or a process currently performing

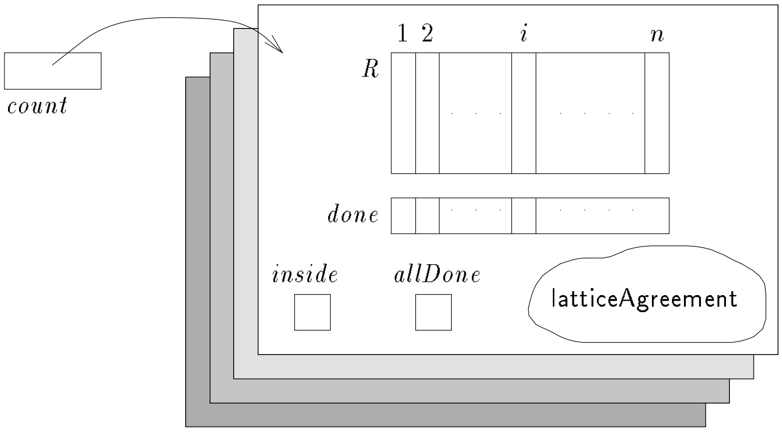


Figure 3: Data structures for a sieve: *count* indicates the current copy; previous copies are shadowed.

getName), or overlaps such a process. Thus, it can be shown that the number of sieves a process accesses is bounded by a linear function of the number of simultaneously participating processes. This allows to bound the space of new names and the step complexity.

For each sieve there is an integer variable, *count*, indicating the current copy of the sieve; it is initialized to 1. There is an infinite number of copies for each sieve, numbered $1, 2, \ldots$. Each copy has the following data structures (see Figure 3):

– An array $R[1, \ldots, n]$ of views; all views are initially empty. Entry $R[i]$ contains the view obtained by process p_i in this copy.

– An array $done[1, \ldots, n]$ of Boolean variables; all entries are initially **false**. Entry $done[i]$ indicates whether process p_i is done with this copy of the sieve.

– A Boolean variable *allDone*, initially **false**. Indicates whether all processes which could get a name from this copy are done.

– A Boolean variable *inside*, initially **false**. Indicates whether any process is already inside this copy.

In addition, we associate a separate adaptive procedure for one-shot scan of the participating processes, latticeAgreement, with each copy.

The pseudocode appears in Algorithm 1. The main code for getting and releasing names appears in procedures getName and releaseName; they rely on procedures enter and leave for entering and leaving a sieve. To simplify the pseudocode, we associate a virtual Boolean variable, *allDone* with copy 0 of every sieve, whose value is **true**.

Algorithm 1 Long-lived renaming: code for process p_i.

private variables // persistent between getName and the following releaseName
    c: integer, initially 0
    W: view, initially $\emptyset$

procedure getName() // get a new name
    $s := 0$
    repeat
        s++ ;
        $rank$ = enter($sieve[s]$)
    until ($rank \neq 0$)
    return($\langle s, rank \rangle$)

procedure releaseName($\langle s, rank \rangle$) // release a name
    leave($sieve[s]$)

procedure enter($sieve$) // enter a specific sieve
    $c = sieve.count$ // locate current copy
    if $sieve.allDone[c-1]$ and (not $sieve.inside[c]$) then
        // candidates of previous copy are done and current copy is free
        $sieve.inside[c]$ = **true**
        V = $sieve$.latticeAgreement$[c]$() // adaptive single-shot scan of participants, from [3]
        $sieve.R[c][i] = V$
        W = candidates($sieve$, c)
        if ($p_i \in W$) then // p_i is a winner in copy c
                $sieve.count = c + 1$
                return the rank of id_i in W
        $sieve.done[c][i]$ = **true** // not a winner in this sieve
        W = candidates($sieve$, c)
        leave($sieve[s]$)
    return 0

procedure candidates($sieve$, $copy$) // returns the possible winners in a copy of the sieve
    $V = sieve.R[copy][i]$ // your view
    $W = \min\{sieve.R[copy][j] \mid p_j \in V$ and $sieve.R[copy][j] \neq \emptyset\}$ // minimal by containment
    if for every $p_j \in W$, $sieve.R[copy][j] \supseteq W$ then return W
    else return $\emptyset$

procedure leave($sieve$) // leave a sieve, c and W remain from the last enter
    $sieve.done[c][i]$ = **true**
    if for every $p_j \in W$, $sieve.done[c][j]$ then
        $sieve.allDone[c]$ = **true** // all candidates of this copy are done

4 Proof of Correctness

First, we prove properties that hold for every copy of every sieve; to do so, fix copy c of sieve s. Note that procedure $\mathsf{candidates}(s, c)$ returns either an empty view (in Line 4) or a non-empty view (in Line 3). The next lemma proves that all non-empty views returned by procedure $\mathsf{candidates}$ for copy c of sieve s are identical.

Lemma 4.1 *If W_1 and W_2 are non-empty views returned by invocations of $\mathsf{candidates}(s, c)$ then $W_1 = W_2$.*

Proof: Let S be the minimal view (by cardinality) obtained in $\mathsf{latticeAgreement}[c]$ of sieve s, by some process p_k. We argue that any non-empty view W returned in the invocation of $\mathsf{candidates}(s, c)$ is equal to S.

Assume, by way of contradiction, that an invocation of $\mathsf{candidates}(s, c)$ by process p_i returns $W \neq S$. Since W is non-empty it is a view obtained in some invocation of $\mathsf{latticeAgreement}[c]$; therefore, W and S are comparable. By the minimality of S, $S \subset W$. Since $p_k \in S \subset W$, p_i checks $R[k]$ in Line 3. If $R[k]$ is empty then p_i clearly fails the test in Line 3; otherwise, $R[k] = S$ and since $S \subset W$, p_i fails the test in Line 3. In both cases, p_i returns $\emptyset$, which is contradiction. ∎

Process p_i is a *winner* in copy c of sieve s if it returns from sieve s with $rank \neq 0$. Clearly, a winner (in some copy) of sieve s gets a name based on its rank in this sieve. By the code, a winner obtains a non-empty view from $\mathsf{candidates}$, containing itself. Lemma 4.1 implies:

Lemma 4.2 *If p_i is a winner in copy c of sieve s, then p_w appears in every non-empty view returned by an invocation of $\mathsf{candidates}(s, c)$.*

Process p_i is *inside* copy c of sieve s after it executes Line 3 of enter with c (after reading **false** from $inside[c]$). A process inside copy c of sieve s is *done* after it assigns 1 to $done[c][i]$ (in Line 1 of $\mathsf{leave}(s, c)$, if it is a winner, or in Line 10 of $\mathsf{enter}(s, c)$, otherwise).

The next lemma shows that winners are only inside a single copy of a specific sieve.

Lemma 4.3 *If process p_i is inside copy c of sieve s, then all winners of smaller copies, $1, \ldots, c-1$, of sieve s are done.*

Proof: We prove the lemma by induction on c. The base case, $c = 1$, is trivial by the initialization.

For the induction step, assume the lemma holds for copy c, and we prove it holds for copy $c+1$. Only a winner in copy c can write $c+1$ to *count*. Therefore, if some process p_i is inside copy $c+1$, it follows that some process p_j is a winner in copy c. In particular, p_j gets inside copy c, and by the induction hypothesis, all winners in copies $1, \ldots, c-1$ are done.

We now prove that the winners of copy c are also done. Since p_i is inside copy $c+1$, it reads **true** from $allDone[c]$ in Line 2. This implies that there is a winner, p_w, in copy c which sets $allDone[c]$ to **true**, in Line 3 of leave. Let W be the view obtained by p_w in candidates(s, c). Since $W \neq \emptyset$, Lemma 4.2 implies that all winners of copy c are in W. By the code, p_w reads 1 from $done[c][j]$, for every $p_j \in W$, before it sets $allDone[c]$ to **true**; therefore, all winners of copy c are done when p_i is inside copy $c+1$. ∎

Only winners get a name from a sieve. At any time, there are winners only in a single copy of this sieve (by Lemma 4.3). Since they have non-empty views, Lemma 4.1 implies that all of them have the same view in this copy. Hence, processes which get their names from the same sieve must have different ranks in this view. This implies that our algorithm satisfies the uniqueness property.

Corollary 4.4 *No two processes hold the same name at the end of a finite prefix of the execution.*

The rest of the proof shows that the name space and the step complexity of the algorithm are truly adaptive.

An execution interval of getName by some process p_i, β, can be partitioned into disjoint intervals during which p_i accesses different sieves. If p_i accesses sieve s, then $\beta|_s$ is the interval of β which starts when p_i executes the first line of enter for sieve s (or the beginning of β, if $s = 1$) and ends when p_i executes the first line of enter for sieve $s+1$ (or the end of β, if p_i wins sieve s). Thus, β can be written as $\beta|_1\beta|_2 \ldots$.

The next lemma shows that processes must access a specific copy of a sieve *simultaneously*, and can not do so at different points.

Lemma 4.5 *If p_i goes inside copy c of sieve s in β_i and p_j goes inside copy c of sieve s in β_j then $\beta_i|_s$ overlaps $\beta_j|_s$.*

Proof: Both p_i and p_j read **false** from $inside[c]$ (in Line 2 of enter) and then write **true** to $inside[c]$ (in Line 3 of enter). Thus, both p_i and p_j read before the first of them writes (see Figure 4), and the execution of Lines 2–3 by p_i and p_j (for copy c of sieve s) must overlap. ∎

We show that at least one of the processes which get inside a copy of a sieve is a winner.

Lemma 4.6 *At least one of the processes which assign true to $s.inside[c]$ is a winner of copy c of sieve s.*

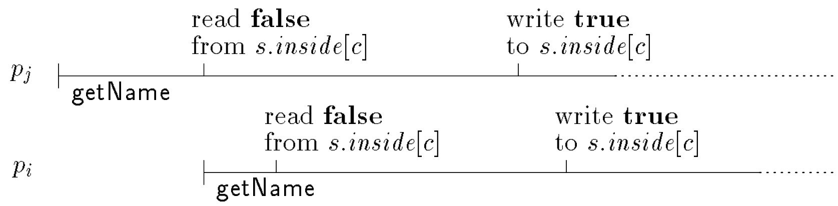


Figure 4: Illustration for the proof of Lemma 4.5.

Proof: Let S be the minimal (by containment) view obtained from latticeAgreement$[c]$ (in Line 4) by some process p_j. This view exists since at least one process reads **false** from *inside*$[c]$ (in Line 2). Let $p_w \in S$ be the last process to write its view to $R[c][w]$, and let V_w be the view obtained by p_w from latticeAgreement$[c]$ (in Line 4). Note that p_j and p_w can be the same process.

Since S and V_w are comparable and since S is minimal, $p_j \in S \subseteq V_w$; moreover, p_j writes its view to $R[c][j]$ no later than p_w writes its view to $R[c][w]$ (in Line 5 of enter) and therefore, before p_w calls candidates (in Line 6 of enter). Therefore, p_w reads S from $R[c][j]$ in Line 2 of candidates, and sets W_w to S, by the minimality of S.

Since p_w is the last to write, it performs Line 3 of candidates after every process $p_i \in W_w = S$ wrote its view V_i to $R[c][i]$. By the minimality of S, $W_w = S \subseteq V_i$, for every $p_i \in W_w$, and therefore, p_w evaluates the condition in Line 3 to be true. That is, p_w is a winner in sieve s. By the code, p_w assigns **true** to *inside*$[c]$ before participating in latticeAgreement$[c]$. ■

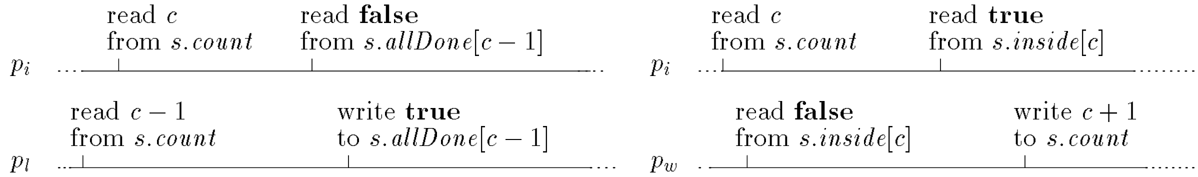
By Lemma 4.2, a non-empty view returned by candidates includes all winners. It is possible that p_i obtains an empty view from candidates and does not see itself as a winner, yet later processes will see p_i in a non-empty view obtained from candidates. This is why a process leaving a sieve sets *done* to 1 even if it is not a winner.

Process p_i is a *candidate* in copy c of sieve s if there is an invocation of candidates(c, s) which returns view W, such that $p_i \in W$. The next lemma shows that a process does not win a sieve only if some concurrent process is a candidate in this sieve.

Lemma 4.7 *If p_i accesses sieve s and does not win, then there is a candidate of sieve s which is inside sieve s concurrently with p_i.*

Proof: Let c be the value read by p_i from $s.count$.

Assume p_i evaluates the condition in Line 2 of enter to **true**. Then p_i assigns **true** to $s.inside[c]$. By Lemma 4.5, all processes which assign **true** to $s.inside[c]$ are concurrent. By Lemma 4.6, at least one of them is a winner in copy c of sieve s.



(a) $allDone[c-1]$ is **false** (b) $inside[c]$ is **true**

Figure 5: Illustration for the proof of Lemma 4.7.

Assume p_i evaluates the condition in Line 2 of enter to **false**. There are two cases: Either $s.allDone[c-1]$ is **false** or $s.inside[c]$ is **true**.

Case 1: p_i reads **false** from $s.allDone[c-1]$. We argue that when all candidates of copy $c-1$ leave the sieve, $s.allDone[c-1]$ is set to **true**. Thus, when p_i reads **false** from $s.allDone[c-1]$, there is at least one candidate of copy $c-1$ inside the sieve.

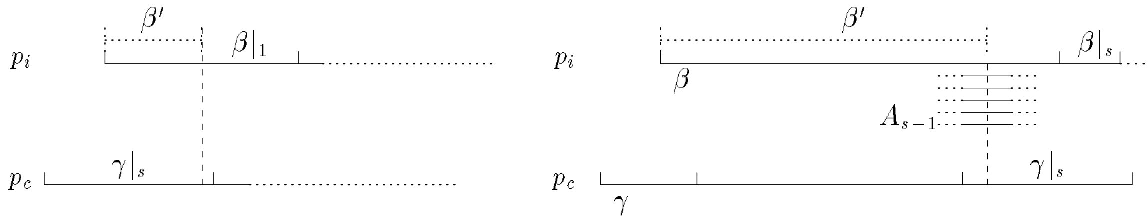
Let W_{c-1} be the set of candidates in copy $c-1$. W_{c-1} is not empty, since there is a process which writes c into $s.count$. Let p_l be the last process in W_{c-1} which writes **true** to its variable $s.done[c-1][l]$ (in line 1 of leave, if p_l is a winner, or in line 10 of enter otherwise). It can be seen that p_l computes the condition in line 2 of leave to be **true**, and sets $s.allDone[c-1]$ to be **true**. Therefore, p_l writes **true** to $s.allDone[c-1]$ after p_i reads it (Figure 5(a)). By Lemma 4.3, p_l reads $c-1$ from $s.count$ before p_i reads c from $s.count$. Thus, p_i is in sieve s concurrently with p_l, a candidate in copy $c-1$ of sieve s.

Case 2: p_i reads **true** from $s.inside[c]$. Thus, there is a process which sets $s.inside[c]$ to **true**. By Lemma 4.6, at least one of the processes setting $s.inside[c]$ to **true** is a winner in copy c of sieve s; call this process p_w. Process p_w reads **false** from $s.inside[c]$ before p_i reads **true** from $s.inside[c]$. Additionally, p_w sets $s.count$ to $c+1$ after p_i reads c from $s.count$, by Lemma 4.3. Thus, p_i is concurrent with p_w, a winner of copy c of sieve s (Figure 5(b)). ■

For the rest of the analysis, let β be an execution interval of getName by process p_i and let β' be a prefix of β. Denote by $A_s(\beta')$ the set of processes accessing sieves $1,\ldots,s$ at the end of β' (possibly different sieves). Denote by $W_s(\beta')$ the set of winners in sieves $1,\ldots,s$ whose execution interval cover the end of β' (possibly different sieves). Note that A_s and W_s are not necessary disjoint.

The next lemma is the key to showing that the name space and the step complexity of the algorithm are truly adaptive.

Lemma 4.8 *Assume that p_i accesses sieve s and does not win. Then there is a prefix β' of $\beta|_1\beta|_2\ldots\beta|_s$, such that $|W_s(\beta')|+|A_s(\beta')|\geq s+1$.*



(a) Base case.

(b) Simple case in the induction step.

Figure 6: Illustration for Lemma 4.8.

Proof: The proof is by induction on s.

The base case is $s = 1$. By Lemma 4.7 there is a candidate p_c in sieve 1 such that $\beta|_1$ overlaps $\gamma|_1$, where γ is p_c's interval. Thus, there is a prefix β' of $\beta|_1$ such that $\gamma|_1$ covers the end of β'. Both p_c and p_i are accessing sieve 1 at the end of β', that is, $p_c, p_i \in A_1(\beta')$ (Figure 6(a)). Therefore, $|A_1(\beta')| \geq 2$ and the lemma holds.

For the induction step, assume the lemma holds for sieve $s-1$. That is, there is a prefix β' of $\beta|_1\beta|_2\ldots\beta|_{s-1}$, such that

$$|W_{s-1}(\beta')| + |A_{s-1}(\beta')| \geq (s-1)+1 = s.$$

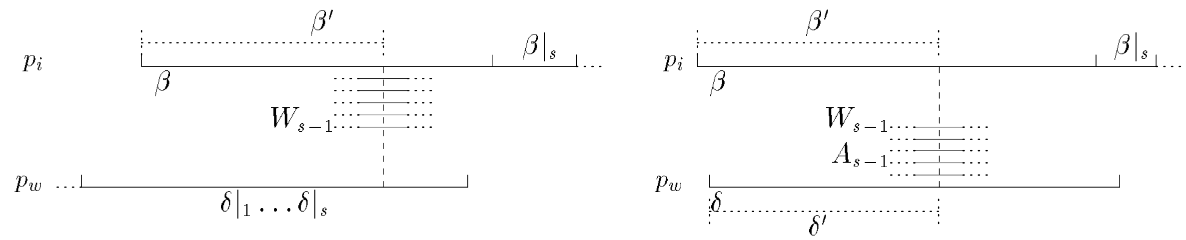
Lemma 4.7 implies that there is a candidate in sieve s, p_c, which is inside sieve s concurrently with p_i. Let γ be p_c's interval; note that $\gamma|_s$ overlaps $\beta|_s$.

First, suppose that $\gamma|_s$ covers the end of β' (Figure 6(b)). By definition, $A_{s-1}(\beta')$ contains only processes which are accessing sieves $1\ldots s-1$ at the end of β'; thus, $p_c \notin A_{s-1}(\beta')$. Since $\gamma|_s$ covers the end of β', p_c is in sieve s at the end of β', that is, $p_c \in A_s(\beta')$. Therefore, $A_s(\beta') \supseteq A_{s-1}(\beta')\dot{\bigcup}\{p_c\}$. Since $W_s(\beta') \supseteq W_{s-1}(\beta')$, we have

$$|W_s(\beta')| + |A_s(\beta')| \geq |W_{s-1}(\beta')| + |A_{s-1}(\beta')| + 1 \geq s+1,$$

by the induction hypothesis.

Otherwise, we argue that there is a winner p_w with interval δ, such that $\delta|_s$ overlaps the interval between the end of β' and the end of $\beta|_s$. If p_c is a winner in sieve s, then it is the desired process p_w. Otherwise, Lemma 4.6 implies that there is a winner p_w in copy c of sieve s; let δ be p_w's interval. If $\delta|_s$ starts after $\beta|_s$ ends, then p_w reads **true** from $s.inside[c]$ and does not go inside sieve s; therefore, $\delta|_s$ does not start after $\beta|_s$ ends. By Lemma 4.5, $\delta|_s$ overlaps $\gamma|_s$. Since $\gamma|_s$ starts after β', it follows that $\delta|_s$ overlaps the interval between the end of β' and the end of $\beta|_s$.



(a) δ starts before β starts. (b) δ starts after β starts.

Figure 7: Illustration for the final cases in the proof of Lemma 4.8.

We consider two cases:

Case 1: δ starts before β starts (Figure 7 (a)). Since p_w wins sieve s, it is not in $W_{s-1}(\beta')$. On the other hand, $\delta|_1 \ldots \delta|_s$ covers the end of β', and hence, $p_w \in W_s(\beta')$. Therefore, $W_s(\beta') \supseteq W_{s-1}(\beta') \dot{\bigcup} \{p_w\}$. Since $A_s(\beta') \supseteq A_{s-1}(\beta')$, we have

$$|W_s(\beta')| + |A_s(\beta')| \geq |W_{s-1}(\beta')| + 1 + |A_{s-1}(\beta')| \geq s + 1,$$

by the induction hypothesis.

Case 2: δ starts after β starts (Figure 7 (b)). Since p_w accesses sieve $s-1$ and does not win, the induction hypothesis for p_w implies that there is a prefix δ' of $\delta|_1 \ldots \delta|_{s-1}$, such that

$$|W_{s-1}(\delta')| + |A_{s-1}(\delta')| \geq (s-1) + 1 = s.$$

Since δ starts after β starts, there is a prefix β' of β that ends with δ'. Clearly, $W_{s-1}(\delta') = W_{s-1}(\beta')$ and $A_{s-1}(\delta') = A_{s-1}(\beta')$. Since p_w wins sieve s, p_w is not in $W_{s-1}(\beta')$; on the other hand, $p_w \in W_s(\beta')$. Therefore, $W_s(\beta') \supseteq W_{s-1}(\beta') \dot{\bigcup} \{p_w\}$. Since $A_s(\beta'') \supseteq A_{s-1}(\beta'')$, we have

$$|W_s(\beta')| + |A_s(\beta')| \geq |W_{s-1}(\beta')| + 1 + |A_{s-1}(\beta'')| \geq s + 1,$$

by the induction hypothesis. ■

Denote $k = \text{Cont}\#(\beta)$. Suppose that p_i does not win in sieves $1 \ldots s$. The previous lemma implies that for some prefix β' of β, $|W_s(\beta')| + |A_s(\beta')| \geq s+1$. Since $W_s(\beta')$ and $A_s(\beta')$ contain simultaneously concurrent processes and since $p_i \notin W_s(\beta')$, we have $|W_s(\beta')| + |A_s(\beta')| \leq (k-1) + k$. Therefore, $s \leq 2k-2$, and p_i must win sieve $2k-1$, if it fails to win a sieve with a smaller number. The rank of p_i sieve s is not greater than k. Thus, we have the following corollary:

Corollary 4.9 *Algorithm 1 assigns names in the range* $\{1, \ldots, (2k-1)k\}$.

Finally, we calculate the step complexity of the algorithm. Suppose that p_i accesses copy c of sieve s in β, and let A be the set of processes which go inside copy c of sieve s throughout the execution, including p_i itself. By Lemma 4.5, processes in A access copy c of sieve s concurrently, implying that $|A| \leq \mathrm{Cont}\#(\beta)$, that is:

Lemma 4.10 *If p_i is inside copy c of sieve s then at most k processes go inside this copy throughout the execution.*

By Lemma 4.8 and the discussion after it, p_i accesses at most $2k-1$ sieves. Clearly, if it does not go inside a sieve (i.e., executes Lines 3–12) then it performs a constant number of steps. Otherwise, the step complexity is determined by the number of processes performing latticeAgreement algorithm in this copy. If p_i is inside a sieve s, then at most k processes perform latticeAgreement for sieve s, by Lemma 4.10; hence, the step complexity of each latticeAgreement is $O(k \log k)$, using the lattice agreement algorithm of Attiya and Fouren [3]. Since the maximal size of a view is k, the step complexity of candidates is $O(k)$.

By the above analysis, a process enters at most k sieves, and by the code, it performs at most a single copy of some sieve. Therefore, the total step complexity of a getName operation is $O(k^2 \log k)$. The total step complexity of the matching releaseName operation is linear in the size of the view W, and thus, it is $O(k)$.

5 Reducing the Name Space

We describe an adaptive algorithm for long-lived $(6k-1)$-renaming which is based on the adaptive k^2-renaming algorithm presented earlier.

A naive way to reduce the name space, which does not work, is to run a long-lived $(2k-1)$-renaming algorithm with the final name obtained in Algorithm 1 as input name. However, existing long-lived $(2k-1)$-renaming algorithms require to read information from all processes; the only known way to do so depends on the range of input names. Since the size of the name space provided by Algorithm 1 varies between 1 and N^2 (depending on the contention, k), the step complexity of the naive algorithm depends on N.

Instead, we follow the adaptive algorithm for one-time renaming [3]. During the execution, participating processes are dynamically divided into sets $G_0, G_1, \ldots G_{\lceil \log n \rceil + 1}$; a disjoint name space of size $2 \cdot 2^j - 1$ is associated with each set G_j. The name spaces are chosen so that the range of final names is linear in k, the number of processes participating simultaneously. A process tries to obtain a name in the space associated with G_0; if it does not succeed, then it tries G_1, etc. In this way, a process moves from set to set; thus, the size of a particular set changes dynamically between 0 and k.

In more detail, the algorithm uses $\lceil \log n \rceil + 1$ identical copies of Algorithm 1, denoted $\mathsf{adaptiveRenaming}_0$, $\mathsf{adaptiveRenaming}_1, \ldots$. It also uses $\lceil \log n \rceil + 1$ copies of the $(2k-1)$-renaming algorithm of Burns and Peterson [8], denoted $\mathsf{BPrenaming}_0, \mathsf{BPrenaming}_1, \ldots$. The

Algorithm 2 Long-lived $O(k)$-renaming (outline).

procedure getName() // $6k-1$ renaming
$j = 0$ // start from G_0
while **true** do // try in set G_j
  $x = \mathsf{adaptiveRenaming}_j(id_i)$ // Algorithm 1
  if $x \leq (2^j)^2$ then
    $y = \mathsf{BPrenaming}_j(x)$ // from [8]
    if $y \leq 2 \cdot 2^j - 1$ then return $2^{j+1} + y$ // obtained a name
  release temporary names y and x
  $j{+}{+}$ // move to the next set

copies of $(2k-1)$-renaming are different: $\mathsf{BPrenaming}_j$ is designed for initial name space $\{1, \ldots, (2^j)^2\}$.

Process p_i attempts to get a name in the space associated with G_j in two stages: In the first stage, p_i performs $\mathsf{adaptiveRenaming}_j$ and obtains a temporary name x. If $x \leq (2^j)^2$, then it belongs to the initial name space of $\mathsf{BPrenaming}_j$; in this case, p_i performs $\mathsf{BPrenaming}_j$ with initial name x, obtaining a new name y. If $y \leq 2 \cdot 2^j - 1$, then it is in the space associated with G_j, and p_i returns the name $2^{j+1} + y$. If the name obtained in the first stage is greater than $(2^j)^2$, or the name obtained in the second stage is greater than $2 \cdot 2^j - 1$, then p_i continues to the next set. The pseudocode is outlined in Algorithm 2.

Lemma 5.1 *Let k be the maximum contention during an execution interval of* getName *by p_i. Then p_i obtains a name in a set G_j, $0 \leq j \leq \lceil \log k \rceil$.*

Sketch of proof: We show that p_i gets a name in the space associated with $G_{\lceil \log k \rceil}$, unless it returns earlier.

By Corollary 4.9, the temporary name obtained in $\mathsf{adaptiveRenaming}_{\lceil \log k \rceil}$ is smaller than $k^2 \leq (2^{\lceil \log k \rceil})^2$. Hence, by the algorithm, p_i proceeds to the second stage. In the $(2k-1)$-renaming algorithm, p_i obtains a name $\leq 2k - 1 \leq 2 \cdot 2^{\lceil \log k \rceil} - 1$ and decides on final name in the name space associated with $G_{\lceil \log k \rceil}$. ■

The rest of the proof is similar to the corresponding algorithm in [3], and is left to the full version of the paper. The step complexity depends on the step complexity of the Burns and Peterson algorithm [8], which is a function of the number of participating processes and the maximal range of names. In every set G_j, both figures depend only on k.

6 Discussion

This paper shows that algorithms can adapt to the actual contention, using only read and write operations. We have presented an adaptive algorithm for long-lived renaming in which process p_i obtains a new name in the range $\{1, \ldots, k(2k-1)\}$ within $O(k^2 \log k)$ steps; k is the maximal number of processes simultaneously participating with p_i. Appendix 5 outlines how to reduce the range of names to $\{1, \ldots, 6k-1\}$.

The obvious open problems are to improve our algorithms by shrinking the range of names, reducing the step complexity, or, perhaps most interestingly, bounding the memory requirements. In another direction, it would be very interesting to find truly adaptive algorithms for other problems, e.g., collecting information and obtaining snapshots, or to prove that they do not exist. Alternatively, it could be worthwhile to explore weaker notions of adaptiveness, e.g., having step complexity which depends on the *total* number of overlapping operations.

Acknowledgments: We thank Yehuda Afek and Eli Gafni for helpful discussions.

References

[1] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 538–547, 1995.

[2] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, July 1990.

[3] H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 277–286, 1998.

[4] A. Bar-Noy and D. Dolev. A partial equivalence between shared-memory and message-passing in an asynchronous fail-stop distributed environment. *Mathematical Systems Theory*, 26(1):21–39, 1993.

[5] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 91–100, 1993.

[6] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 41–52, 1993.

[7] E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG distributed simulation algorithm. Technical Report MIT/LCS/TM-573, Laboratory for Computer Science, MIT, Dec. 1997.

[8] J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 145–158, 1989.

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

[10] A. Fouren. Wait-free algorithms with contention-dependent complexity. Master's thesis, Department of Computer Science, The Technion, Mar. 1998.

[11] M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[12] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, Feb. 1987.

[13] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Programming*, 25(1):1–39, Oct. 1995.

[14] M. Moir and J. A. Garay. Fast long-lived renaming improved and simplified. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 1996.