Adaptive algorithms utilyzing adaptive collect and snapshot

Yehuda Afek* Eli Gafni† Michael Merritt‡

August 25, 1999

Abstract

Several adaptive algorithms are automatically generated via a simple transformation from single-writer multi-reader algorithms, using the $O(k)$ adaptive collect algorithm of Attiya and Fouren [AF98a]. Among these algorithms are an adaptive snapshot algorithm with step complexity $O(k^2)$, and three algorithms solving $(2k-1)$-renaming, but with high step complexities ($k4^k$, $\Omega(2^{k/2})$, and k^3), where k is the contention, the number of processes actually taking steps during the run of the algorithm. The transformation does not always produce an adaptive algorithm: the $O(n \log n)$ lattice-agreement (one shot snapshot) algorithm of Attiya and Rachman [AR93] is one example. However, we show that a simple modification of the original algorithm allows the transformation to produce an adaptive, $O(k \log k)$ lattice-agreement algorithm, matching the best-known step complexity of Attiya and Fouren's algorithm [AF98].

Finally, we present a specially-tailored algorithm for $(2k-1)$-renaming that uses any of these adaptive lattice-agreement or snapshot algorithms as a component. Building on a renaming algorithm of Attiya and Fouren, the adaptive $(2k-1)$-renaming algorithm has step complexity $O(k^2)$, which is better than any other known adaptive algorithm for optimal renaming.

1 Introduction

Since early work in mutual exclusion [Lam87], researchers have asked whether distributed algorithms can be made *adaptive*; that is, can their worst-case step complexity be bounded by a function of the number, k, of actually active or contending processes, rather than the total number of processes, N, that *might* take steps [ADT95]? (We also use n to denote an *a priori* known upper bound on k, which in the worst case is N.)

In [AF98] Attiya and Fouren provide several adaptive constructions for wait-free collect, lattice-agreement and renaming. Lattice-agreement is a one-shot version of atomic snapshot. It is a linearizable decision-problem in which each process *writes* its name, and then performs a *snapshot* operation, which returns the subset of processes which have previously written. In the K-renaming problem, as many as (but no more than) n processes execute *rename* operations. They begin with unique names taken from a set of size N, and exit with unique names from a set of size $K < N$. For

*Tel Aviv University, Tel-Aviv, Israel. afek@math.tau.ac.il. Part of this work was performed while visiting at AT&T Labs.

†U.C.L.A., Los-Angeles, CA 90024. eli@cs.ucla.edu

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

wait-free algorithms utilizing read-write registers, K must be at least $2n-1$ [HS93]. Previous work describes a non-adaptive algorithm meeting this bound [ABND$^+$90]. Among Attiya and Fouren's adaptive solutions to the renaming problem, the one with the smallest output name space reduces the name space arbitrarily close to $4k-1$ names, where k is the number of active processes. This leaves open the question whether a sub-optimal name space is an inevitable price of utilizing an adaptive algorithm. Our work closes the remaining gap, demonstrating that an optimal name space of $2k-1$ can be achieved with an adaptive algorithm whose step complexity is $O(k^2)$.

In this paper we first use the adaptive collect of Attiya and Foren to describe a simple method to transform algorithms that use single-writer multi-reader (SWMR) shared-memory into algorithms that use multi-writer multi-reader (MWMR) shared-memory. Essentially, the transformation replaces reads in the original algorithm with adaptive collect operations. As we show, in certain cases the transformed algorithm is adaptive. Examples of algorithms that become adaptive are the $(2k-1)$-renaming algorithms of Bar-Noy and Dolev [BD89], Burns and Peterson [BP89], and Gafni [G92]. The transformed adaptive $2k-1$-renaming algorithms have relatively high step complexities. (The smallest is $O(k^3)$.)

The transformation does not always produce adaptive algorithms: applying it to Attiya and Rachman's lattice-agreement algorithm [AR93] and to the Borowsky and Gafni immediate snapshot algorithm [BG93] fails to produce corresponding adaptive algorithms. This is because these algorithms depend on N or n, bounds on the total number of processes or on the maximum number of participants, respectively, in intrinsic ways. Sometimes the original algorithm can be modified to remove this intrinsic dependancy. As an example, in Section 3 we present a modification of the Attiya and Rachman lattice-agreement algorithm whose complexity is $O(N \log k)$. Applying the transformation to the modified algorithm automatically produces an adaptive $O(k \log k)$ lattice-agreement algorithm, matching the complexity of the best known adaptive solution [AF98].

Furthermore, the transformation does not always produce efficient adaptive algorithms: this is the case in the three applications to the $(2k-1)$-renaming algorithms described above. In Section 4 we describe a more efficient adaptive $(2k-1)$-renaming algorithm.

First, we describe an algorithm for renaming processes with input names from a space of size $f(k)$ to output names in a space of size $2k-1$, in time $O(f(k)^2)$, where $f(k)$ is a bound on the largest input name among any k active processes.[1] This algorithm is a modification of another algorithm by Attiya and Fouren, which solves $(2k-1)$-renaming but has $O(N)$ worst case adaptive time complexity [AF98], where N again is a bound on the original name space. This worst-case complexity arises even in runs in which the largest input name of an active process is very small relative to N.

Given this algorithm for reducing names from a space of size $f(k)$ to $2k-1$, in time $O(f(k)^2)$, we can first run, for example, the adaptive Attiya-Fouren algorithm for $6k-1$-renaming in time $O(k \log k)$ [AF98], so that $f(k) = 6k-1$. The result is the most efficient known adaptive $(2k-1)$-renaming algorithm, requiring $O(k^2)$ time.

Table 1 summarizes these results, listing the algorithms presented in this paper, previously-known adaptive algorithms for these problems, and the building blocks used in the constructions. Our work reinforces Attiya and Fouren's elegant demonstration that simple algorithmic tools can be combined in powerful ways. Focusing specifically on adaptive building blocks and techniques, we develop and enrich the set of known adaptive algorithms.

[1] An extended abstract describing this algorithm appeared in [AM99].

Problem	Asymptotic step complexity	Adaptive?	Reference
Collect	k	Yes	[AF98a]
$(2k-1)$-renaming	$n4^n$		[BD89]
$(2k-1)$-renaming	$\Omega(2^{n/2})$		[BP89, F99]
$(2k-1)$-renaming	n^3		[G92]
$(2k-1)$-renaming	N		[AF98]
$(4k-1)$-renaming	$k \log k$	Yes	[AF98]
$(2k-1)$-renaming	$k4^k$	Yes	This paper.
$(2k-1)$-renaming	$\Omega(2^{k/2})$	Yes	This paper.
$(2k-1)$-renaming	k^3	Yes	This paper.
$(2k-1)$-renaming	k^2	Yes	This paper.
lattice-agreement	$N \log n$		[AR93]
lattice-agreement	$k \log k$	Yes	[AF98]
lattice-agreement	$k \log k$	Yes	This paper.
Snapshot	kN		[AADGMS]
Snapshot	k^2	Yes	This paper.

Table 1: Algorithms discussed in this paper

2 Automatically constructing adaptive algorithms

In this short section we describe a simple method for transforming one-shot algorithms utilizing single-writer multi-reader (SWMR) shared-memory into algorithms using multi-writer multi-reader (MWMR) shared-memory. As we demonstrate, for many SWMR algorithms, the resulting MWMR algorithm is adaptive.

To apply the transformation, we assume the SWMR algorithm takes the simple, natural form in which each process p_i alternates between writing its own variable V_i and reading all the variables $V_1, \ldots, V_n$. The transformation simply uses the $O(k)$, adaptive write and collect procedures of Attiya and Fouren [AF98a] in place of each successive write and n reads of the SWMR algorithms, respectively.

2.1 Adaptive algorithms produced by the transformations

For many SWMR algorithms, this transformation removes any dependence of the step complexity on n, resulting in adaptive MWMR algorithms. Examples of such algorithms for renaming include Bar-Noy and Dolev's $O(n4^n)$ solution [BD89], the algorithm of Burns and Peterson [BP89], which has $\Omega(2^{n/2})$ step complexity [F99], and Gafni's $O(n^3)$ algorithm [G92]. Applying the transformation on this algorithms result in adaptive, wait-free $(2k-1)$-renaming algorithms with worst-case step complexity of $O(k4^k)$, $\Omega(2^{k/2})$, and $O(k^3)$, respectively. The step complexity of each of these algorithms is higher than $O(k^2)$, the step complexity of the specially-tailored adaptive renaming algorithm presented in Section 4.

In addition, the transformation can be applied to the simple $O(kN)$ snapshot algorithm of Afek, *et al.* [AADGMS], resulting in an adaptive $O(k^2)$ snapshot algorithm. The resulting snapshot algorithm supports any number of writes and scan operations, unlike the one-shot variant, lattice-agreement.

2.2 The transformation does not always produce adaptive variants

While many algorithms are made adaptive when the transformation is applied to them, e.g., the renaming algorithms mentioned above, in some algorithms the dependence on n is more involved. In such cases the transformed algorithms are still not adaptive. But in some cases the dependence on n is easily removed, as we show below for the lattice-agreement algorithm of Attiya and Rachman [AF98]. In other algorithms, such as the immediate snapshot algorithm of Borowsky and Gafni [BG93], the dependence on n appears to be more difficult to remove.

In the $O(n \log n)$ lattice-agreement algorithm of Attiya and Rachman, processes traverse a balanced binary tree of depth $\log n$, spending $O(n)$ time at each node, reading and writing shared memory. Replacing these reads and writes with the corresponding collect and write operations via the transformations results in an algorithm with step complexity $O(k \log n)$. However, in this case, the data structure can be re-structured, transforming the original algorithm to a similar algorithm with $n \log k$ complexity. Applying the transformation to the restructured algorithm results in a simple $O(k \log k)$ lattice-agreement algorithm, matching the step complexity of Attiya and Fouren's adaptive lattice-agreement [AF98]. The next section contains the details of the restructuring.

3 A simple $O(n \log k)$ lattice-agreement algorithm

In this section we modify the lattice-agreement algorithm of Attiya and Rachman [AR93] so that the automatic transformation produces an adaptive variant. The Attiya and Rachman algorithm solves lattice-agreement for n processes, using a balanced binary tree of depth $\log n$. (For simplicity, we assume that n is a power of 2.) Our modification uses $\log n + 1$ copies of this algorithm, one for every integer i from 0 to $\log n$, each of size 2^{i+1}. Each process uses at most $O(\log k)$ phases of writing and reading to obtain an estimate on k, and chooses a corresponding copy of the Attiya and Rachman algorithm to execute. This estimate on k itself uses key ideas from the Attiya and Rachman algorithm.

The basic idea of the algorithm is to classify processes that perform lattice-agreement operations into groups whose sizes are bounded by $1, 2, 4, \ldots, 2^i, \ldots, 2^{\log n}$, such that,

1. the lattice-agreement of processes in group i contains no more than 2^i and no less than 2^{i-1} processes,
2. a process that is placed in group i includes in its lattice agreement all the processes that were placed in group j, for all $j < i$, and
3. at least 2^{i-1} processes were active in the lattice-agreement before or concurrently with any process in the i'th group.

Within each group we then use the lattice-agreement algorithm of Attiya and Rachman. The lattice-agreement of processes in group i are linearized before all the lattice-agreements of processes in group j, $j > i$.

The placement in the groups is implemented by processes repeatedly writing and reading (collect) all the processes they know are active. A process that reads and writes a group of size larger than 2^{i-1} and smaller than 2^i three consecutive times places itself in group i and performs lattice-agreement in the i'th copy of the Attiya and Rachman lattice-agreement. Notice that if a process does not see three such consecutive group sizes, then the number of processes it sees doubles, which can happen at most $\log k$ times.

Algorithm 1 Lattice-agreement code for process with input name p.

Function LA(p), for input name p — Performs lattice-agreement for process with input name p.

Shared:

$Mem[1..n]$: array of sets of process indices

$Lattice_{[0..\log n]}()$ — $Lattice_i$ solves lattice-agreement for 2^{i+1} processes,

Local:

C_1^p, C_2^p, C_3^p, s: set of process names

i: integer, $0 \leq j \leq \log n$

j: integer, $1 \leq j \leq n$

$C_3^p := \{p\}$

repeat

  $Mem[p] := C_3^p$ — First such $Mem[p] := C_3^p$ implements L.A. *write* operation.

  $C_1^p := \cup_{j=1}^{N} Mem[j]$

  $Mem[p] := C_1^p$

  $C_2^p := \cup_{j=1}^{N} Mem[j]$

  $Mem[p] := C_2^p$

  $C_3^p := \cup_{j=1}^{N} Mem[j]$

until $\exists i : (2^i \leq |C_1^p| < 2^{i+1}) \wedge (2^i \leq |C_3^p| < 2^{i+1})$

$return(Lattice_{i+1}(C_2^p))$

More specifically, the crux of the Attiya and Rachman algorithm is based on properties of three successive write/read phases, summarized in the Lemma below. That is, processors repeatedly write and read sets of names of other processors, starting with a write of the singleton set containing their name, and each successive write containing the union of all sets read. Consider three successive pairs of writes and reads by process p: process p writes a set C_0^p, then reads all cells, and takes their union, denoted C_1^p. It then writes C_1^p, reads and unions again, obtaining C_2^p, and writes C_2^p. Finally it reads again: denote the union of this set of cells by C_3^p. That is, we have three iterations of write followed by a read, where the second and third writes contain the union of the preceding reads:

$Mem[p] := C_0^p$
$C_1^p := \cup_{j=1}^{N} Mem[j]$
$Mem[p] := C_1^p$
$C_2^p := \cup_{j=1}^{N} Mem[j]$
$Mem[p] := C_2^p$
$C_3^p := \cup_{j=1}^{N} Mem[j]$.

Consider two such sequences, carried out by processes p and q.

Lemma 1 *If $|C_3^p| \leq |C_1^q|$, then $C_2^p \subset C_2^q$.*

Proof: Suppose first that $C_1^q \subset C_3^p$. Then $C_1^q = C_3^p$, and we have $C_2^p \subset C_3^p = C_1^q \subset C_2^q$.

Alternatively, there exists x in C_1^q but not C_3^p. This means that $C_3^p := \cup_{j=1}^{N} Mem[j]$ begins before $Mem[p] := C_1^p$ ends. Hence $Mem[p] := C_2^p$ ends before $C_2^q := \cup_{j=1}^{N} Mem[j]$ begins, and $C_2^p \subset C_2^q$. ■

The adaptive algorithm is now straightforward. Processor p tries to fit three consecutive reads to return sets all with sizes in the same interval $[2^i, 2^{i+1})$, for some non-negative integer i. Once it succeeds, it takes C_2 as input to the Attiya and Rachman algorithm for 2^{i+1} processes.

The details of this algorithm are shown in Figure 1. The correctness of this algorithm is simple to demonstrate, following modularly from the properties of the lattice-agreement components. The first lemma below assures that only the appropriate number of processes (and process names) enter any copy of the *Lattice* algorithm:

Lemma 2 *For $0 \leq i \leq \log n$, at most $2^{i+1} - 1$ distinct processes p call $Lattice_{i+1}(C_2^p)$, and the union of their inputs C_2^p contains at most $2^{i+1} - 1$ values.*

Proof: Let C be the union of the set of inputs to $Lattice_{i+1}()$ in some run of the algorithm in Figure 1, and let p be the last process in C to execute its first write operation, $Mem[p] := \{p\}$. Since $p \in C_2^q$ for some process q that called $Lattice_{i+1}(C_2^q)$, it follows that $\mathrm{C} \subset C_3^q$, and $\mathcal{C} \leq |C_3^q| < 2^{i+1}$. ■

Each lattice-agreement operation consists logically of two distinct operations: the initial write of it's name, and a scan that returns a set of names of active processes. (In calls to $Lattice_i$, the initial write is a set of process names, and the scan returns the union of sets of inputs of active processes.) We must show that all pairs of scans are comparable (one is a subset of the other), and can be ordered together with the writes to preserve read/write semantics and respect the partial "happens-before" order of the lattice operations [Lam78].

Pairs of scans within the same copy of *Lattice* are comparable by the correctness of the Attiya and Rachman algorithm [AR93]. The comparability of pairs of scans from different copies of Lattice follow from Lemma 1.

The ordering of scans which make calls to the same copy of *Lattice* are ordered according to the order of their encapsulated scans, as dictated in the correctness proof of the Attiya and Rachman algorithm [AR93], which ensures this order respects their comparability. All scans which make calls to $Lattice_i$ are ordered before scans which make calls to $Lattice_j$, for all $j > i$. (By Lemma 1, this also respects the comparability of scans.) As in the proof of the Attiya and Rachman algorithm [AR93], writes are ordered after the last scan omitting them, and before the first including them. This certainly respects read/write semantics.

It remains to argue that this ordering respects the happens-before relation. Certainly if a write by p occurs before a call by q to the algorithm begins, or after it ends, then p will either be included or excluded from the scan q returns, so the ordering of writes and scans respects the happens-before relation. The happens-before relation of calls to a single copy of *Lattice* refine the calling operations' happens-before relation, so their ordering also respects the happens-before relation. Finally, Lemma 1 assures that if an operation calling $Lattice_i$ completes before an operation calling $Lattice_j$ begins, then $i < j$, and their order also respects the happens-before relation.

Replacing the reads and writes with adaptive collect operations automatically transforms the algorithm's step complexity from $O(n \log k)$ to $O(k \log k)$.

4 $(2k-1)$-renaming in $O(k^2)$ steps

Our algorithm is an adaptation of Algorithm 2 from Attiya and Fouren [AF98]. The description and analysis in this section is self-contained: Section 5 discusses the relationship between our algorithm and that of Attiya and Fouren in some detail.

4.1 Algorithm overview

Recall that n is an *a priori* bound on the number of active processes in any run. (It is at most N, the bound on the original input name space.) We assume that the original input name space (of

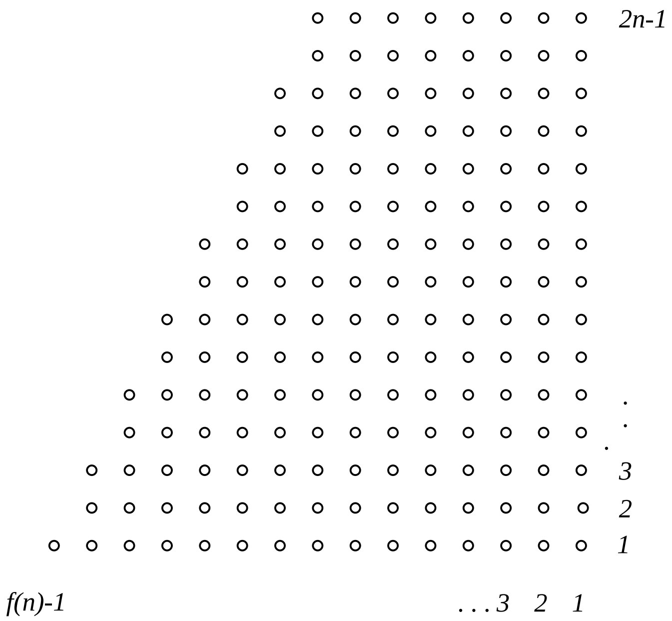

Figure 1: Switch array for $2k-1$-renaming in $O(k^2)$ time. (Switches shown as solid circles are not accessed by the algorithm.)

size N) has been reduced, in any run in which k processes participate, to a space bounded by $f(k)$. Hence, $f(n)$ is in turn a bound on the input name space. (And so $f(n)$ is at most $f(N)$.)

In this algorithm processes flow from left to right through a two-dimensional array, R, of simple shared objects called *switches*. (The corresponding objects are called *reflectors* in [AF98].) The array R has $2n-1$ rows and $f(n)$ columns, where $n \leq N$ is an *a priori* upper bound on k. (See Figure 1.) In the descriptions below, the columns are indexed from $f(n)$ to 1, so that $R[f(n)][1]$ is at the bottom left corner of the array, and $R[1][2n-1]$ is the top right corner. (The switches at and above the diagonal connecting $R[f(n)][1]$ and $R[f(n)-(2n-1)][2n-1]$ are not accessed by the algorithm, and may be omitted.)

Each process acquiring a new name takes a distinct left-to-right path through the array. The left-to-right paths traced by the processes do not intersect each other. If k processes execute the

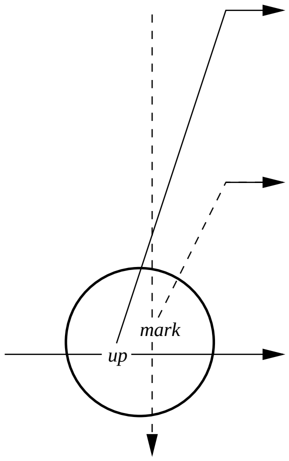

Figure 2: Anatomy of a switch.

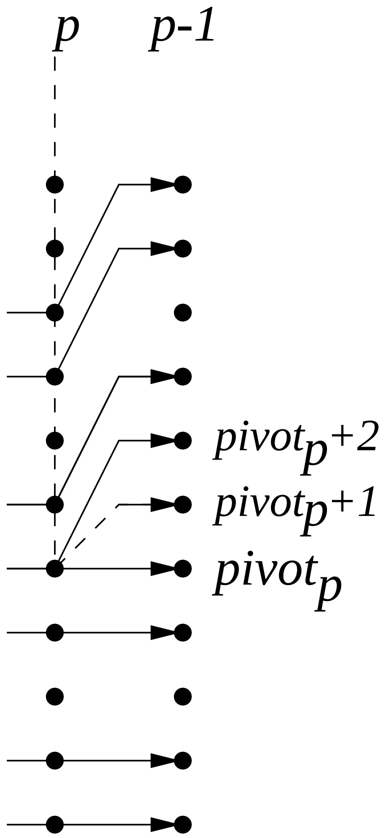

Figure 3: Process movement through column p.

algorithm, each entering with a unique name between 1 and $f(k)$, they descend and are distributed among the bottom $2k-1$ rows of the array: the new name of each process is the row index at which it exits the right-handside of the array.

Each of the switches is implemented by two MWMR bits, *up* and *mark*. The *up* bit's values instruct a process that enters the switch from the left: the default setting indicates it should continue moving horizontally in the same row into the next column of the array, and the other setting redirects it to move up two rows in entering the next column. In addition to this setting each of the switches is either marked or unmarked. Initially the switches are unmarked (*mark* bit set to false) processes mark a switch when they intend to move through it horizontally. (See Figure 2. The solid line shows the process flow from left-to-right, depending on the switch setting. The dotted line shows process p's action depending on whether the switch is marked.)

4.1.1 Actions within a column

Each process enters the algorithm in the distinct column indexed by its input name. When process p enters column p, it descends through the column setting switches up, (until it finds the topmost switch in the column, at a row $pivot_p$, which has been marked by a process intending to move horizontally through it. Process p joins the processes flowing left-to-right in the next column to the right, column $p-1$, one row above $pivot_p$. (See Figure 3.) All the processes that move through column p below $pivot_p$ move horizontally to column $p-1$. Any process that enters column p above row $pivot_p$ is redirected up two rows when it enters column $p-1$, thus making room for p's movement

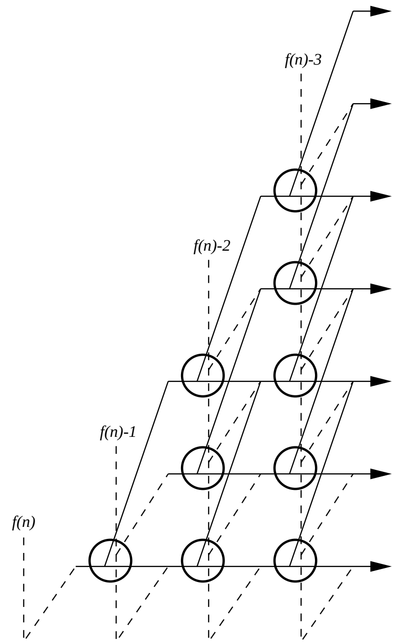

Figure 4: Details of the switch array.

to row $pivot_p$+1 in column $p-1$. At the $pivot_p$ point, race conditions between process p setting the $pivot_p$ switch and the process marking it introduce uncertainty, so that the three rows $pivot_p$, $pivot_p$+1 and $pivot_p$+2 in column $p-1$ are reserved for these two processes: the $pivot_p$ switch in column p acts as a 3-renaming object for these two processes. Figure 4 summarizes the details of how processes may move through the first few columns of the array, and shows how four columns implement $(2k-1)$-renaming for up to four processes.

To achieve this behavior, process p effectively switches all the switches above $pivot_p$ to the up position and leaves all the switches below $pivot_p$ in the default horizontal position. Below, we describe how this is done adaptively, so that no more than $2f(k)$ switches in column p are accessed by p.

As each process moves from left to right through the array, it must determine in each column p whether its current index is above or below $pivot_p$. In the Attiya-Fouren algorithm, each process p enters its column from the top, and sets switches to the up position as it descends until it detects a switch marked by a horizontally-moving process. As the array must be at least $2n-1$ rows in height, any such algorithm is not adaptive. Instead of entering from the top of the array, in our adaptive algorithm process p enters the array at an index bounded by a function of the number of processes it observes to be active. To compute this index, before entering the array, process p performs an adaptive *write* and then an adaptive, one-shot *snapshot* operation, returning a set $snap_p = \{s_m, ..., s_1\}$ of active processes, where $s_m > ... > s_1$. (Either the $k \log k$ lattice-agreement algorith of [AF98] or the k^2 snapshot algorithm in Section 2 can be used.) It then enters column p at the switch with index $2(s_m - p) - 1$. (If $s_m = p$, process p accesses no switches in column p and moves to the bottom switch in column $p-1$. See e.g. column $f(n)$ in Figure 4.) Since s_m is bounded by $f(k)$, there are fewer than $2f(k)$ switches below index $2(s_m - p) - 1$, and in an adaptive algorithm p can access all the switches from this point down in the column.

Because p does not enter column p from the top of the array, the processes entering column p from the left cannot simply access the specific switch at their index to determine whether that switch

Algorithm 2 Code for process with input name p

$switch$:struct of bits up and $mark$, initially false
$column[1:2n]$: array of $switches$

Function Rename(p), for input name p — Returns new name for process with input name p.

Shared:
$R[f(n):1]$: array of $columns$ — Switches above $R[i][2f(n)-(2i-1)]$ are unnecessary.

Local:
$snap$: set of input names
s, e, c, $pivot_p$: input names
j, $index$: integers

$write(p)$ — Record p in a lattice-agreement object and take a
$snap := snapshot$ — snapshot, using an efficient, adaptive algorithm.
$s := max_{e \in snap}\ e$
for $c = p$ **down to 1 do** — For each column from $R[p]$ down to $R[1]$.
  if $c \in snap$ **then**
    $j = min(2n-1, 2(s-c)-1)$ — Set the up bits in column c
    while $(j \geq 1$ **and** $\neg R[c][j].mark)$ **do** — from the $2(s-c)-1$st down to 1,
      $R[c][j].up :=$ true — as long as the $mark$ bit is not set.
      $j := j-1$
    od
  else — Set the $mark$ bits in column c
    $j = 1$ — from the 1st up to $index$,
    while $(j \leq index$ **and** $\neg R[c][j].up)$ **do** — as long as the up bit is not set.
      $R[c][j].mark :=$ true
      $j := j+1$
    od
  if $c = p$ **then** — Decide the index into the next column.
    $pivot_p := \max\{\{0\} \cup \{j | 1 \leq j \leq 2(s-p)-1 \text{ and } R[p][j].mark\}\}$
    $index := pivot_p + 1$
  else if $R[c][index].up$ **then** $index := index + 2$
od
return$(index)$

is above or below $pivot_p$. For example, process q might enter column p at a switch j well above row $2(s_m - p) - 1$. Seeing no other process in the switch, q could conclude it is below $pivot_p$, when instead it is above. In our adaptive algorithm, each process q moving from the left into column p will access some of the switches in column p successively, in one of two ways: If process q observed process p in q's snapshot, $snap_q$, then process q mimics process p's actions in column p. That is, starting with the switch at row $2(s_{m'} - p) - 1$, where $s_{m'}$ is the largest name in $snap_q$, process q descends through the column, setting switches to the up position, and discovering $pivot_p$ just as p would have. Alternatively, if process q did not observe process p in its snapshot $snap_q$, then process q starts at the bottom switch of the column, marking each switch as it ascends, until it either finds $pivot_p$, or reaches its own index in p without finding $pivot_p$ (which must thus be above that index, if at all). A key property shows that each process which descends through the switches in column p starts at a point no lower than the incoming index of any process which ascends in that column. (This is shown in Lemma 5.) It follows that each process acquires sufficient information about $pivot_p$ to determine which index to take in column $p-1$, without accessing more than $f(k)$ switches in column p.

4.1.2 Examples

To illustrate with some simple examples: The process with the highest input name enters the array to the left of all the other processes and starts its left-to-right path at the bottom row. It remains on the bottom row until it reaches a column in which another process has already appropriated that row. Alternatively, a solitary process that begins the algorithm after other processes have terminated will trace out a left-to-right path above the paths of other processes, and exit with a name larger than that of the other processes.

4.1.3 Summary

Summarizing, each process p takes a snapshot, $snap_p$, sets switches up as it descends in column p from $2(s_m - p) - 1$ to the marked switch, at row $pivot_p$ (or to the bottom switch), computes the appropriate index into column $p - 1$, and starts moving to the right. On entering column c with index i from the left, process p either descends from index $2(s_m - p) - 1$ as far as index 1, setting switches up, or ascends from index 1 as far as index i, marking them. In each case process p stops if it finds $pivot_p$: that is, if it observes a marked switch while descending, or a switch set up while ascending. As i is no more than $2k-1$ and s_m is bounded by $f(k)$ (which in turn is at least $2k-1$), process p will access no more than $2f(k)$ switches in each of $f(k)$ columns. The switches themselves are simply implemented with two binary flags, accessed in a single read and write. Hence, if an efficient adaptive lattice-agreement algorithm is used (such as that of [AF98] or in Section 2), the resulting $(2k-1)$-renaming algorithm is adaptive, with step complexity $O(f(k)^2)$.

4.1.4 An alternative implementation

An interesting variant of the $(2k-1)$-renaming algorithm views each column c of the array as an independent, incremental renaming algorithm. We describe this variant briefly below–details of the algorithm and its correctness proof are left as an exercise.

In the l'th stage of the incremental algorithm, $l-1$ processes with unique names between 1 and $2l-3$ must coordinate with another potential new participant with input name c. If c does not participate, the other processes retain their original names, and if c does participate, then all l processes exit with unique names between 1 and $2l-1$. Combining $f(n)$ such incremental renaming objects in sequence results in a full $(2k-1)$-renaming algorithm. The challenge is to implement each incremental renaming object with $O(k)$ step complexity. This can be done with a column of $2f(n)-1$ switches (as in our algorithm) with which we now associate an adaptive collect object [AF98]. Each process that enters the incremental renaming updates the collect object with its name and performs an adaptive *collect* procedure. As in our algorithm, if a process does not observe process c, it tries to mark all the switches from the bottom to its input name. Otherwise, if it does observe c in its collect, it mimiks c's operations as in our algorithm. This variant and its proof are very similar to the algorithm given above.

4.2 Correctness proof

We focus on *complete executions* of the algorithm: finite runs in which every process that carries out a rename operation completes it. Since the algorithm has no unbounded loops and makes calls to registers or to wait-free procedures (to $write(p)$ and *snapshot*), each call will return if the calling process takes steps: the algorithm is wait-free. For the safety properties we consider below, we need consider only finite executions, each of which is a prefix of some finite complete execution. For notational convenience, we identify processes with their unique input name between $f(k)$ and 1.

4.2.1 Small output name space

The first lemma limits the height of indices as processes move from column to column in the array R. The second lemma uses the first inductively to show that the output name space for k processes is bounded by $2k-1$. (The remainder of the correctness argument will focus on showing that the output names are distinct, and on the step complexity.)

Lemma 3 *In any complete execution α,*

- *if process p has index j exiting column $c+1$, then it has index at most $j+2$ exiting column c.*
- *If process p's index j is the maximum index computed by any process exiting column $c+1$, then the process with input name c has index at most $j+1$ exiting column c.*

Proof: The first claim is a simple observation on the code for updating an index (which either leaves the index unchanged or increases it by two in each column, 18). The second claim is a consequence of the code for setting indices for the process with name c, which is one more than the maximum i such that $R[c][i].mark =$ true, which is at most j (Lines 16 and 17). ■

Following Attiya and Fouren [AF98], the next lemma shows that the output name space is bounded by $2k-1$, where k is the number of processes which execute renaming operations.

Lemma 4 (AF98) *Let α be a complete execution in which only the processes with input names $s_k, ..., s_1$ perform renaming operations, where $s_k > ... > s_1$. Then for all j, $k \geq j \geq 1$, processes s_k through s_j exit column s_j and any columns between s_j and s_{j-1} with indices no greater than $2(k-j)+1$.*

Proof: (Note first that when $j=1$, the lemma implies that the processes exit the algorithm with names between 1 and $2(k-1)+1 = 2k-1$.)

The claim is proven by induction on j, from k down to 1. Note first that process s_k was the only process which accessed column s_k. While accessing this column, it set $j = 1$ in line 6, droped immediately to line 15 and computed the index 1 in line 17. Since no process with an input name corresponding to any columns between s_k and s_{k-1} was active, process s_k entered each of these columns from the bottom and exited with index 1.

Given the claim for s_k through s_j, we next prove the claim for the column with index s_{j-1}. By induction, the indices of s_k through s_j exiting columns preceding s_{j-1} are at most $2(k-j)+1$. By part 3 of Lemma 3, their indices exiting column s_{j-1} are at most $2(k-j)+1+2 = 2(k-(j-1))+1$. Also by part 3 of Lemma 3, process s_{j-1} computes index at most $2(k-j)+2 = 2(k-(j-1))$ in column s_{j-1}. Since no process with an input name corresponding to any columns between s_{j-1} and s_{j-2} was active, each of the $k-j+1$ processes s_k through s_{j-1} entered these columns from the bottom, and retained the same index. ■

4.2.2 Estimating an entry point in a column

As discussed above, the processes descending in a column need to know a place to start that is as high as the top-most switch marked by any process ascending in that column, but the index of this position must be a function of k, not e.g. $2n$, as in the Attiya-Fouren algorithm. (Although the previous lemma shows that $2n-1$ can be used to upper bound any such function of k.)

Consider a snapshot $snap = \{s_m, ..., s_1\}$, where $s_m > ... > s_1$, computed by some process p. Note first that no process will mark any *switch* in column s_m. (This follows because only process q with input name s_m or bigger will access *any* switch in column s_m, and if $q \neq s_m$, then q is not in *snap*. Therefore, the snapshot computed by q contains *snap*, and hence q, like s_m, will descend in column s_m. Only ascending processes mark switches.) Hence process s_m will compute index 1 in column s_m (By computing $j = -1$ in Line 6 and skipping all the way down to Line 15). By Part 3 of Lemma 3, in each subsequent column s_m can move up no more than two rows. In column s_{m-1}, only process s_{m-1} can mark any switches, so following the "highest" possible trajectory, no switch in column s_{m-1} can be marked above row $2(s_m - s_{m-1}) - 1$. The lemma below extends this argument inductively to show that no switch in any column s_j is marked in a row above $2(s_m - s_j) - 1$. Hence, p can use the minimum of $2n-1$ and $2(s_m - s_j) - 1$ as a "safe" entry point in column s_j.

Lemma 5 *Let α be a complete execution, and let $snap = \{s_m, ..., s_1\}$ be a snapshot returned in α, where $s_m > ... > s_1$. Then for all j, $m \geq j \geq 1$, no switch is marked in column s_j above row $2(s_m - s_j) - 1$, and the indices computed by processes s_m through s_j on exiting column s_j are each at most $2(s_m - s_j) + 1$.*

Proof: The claim is proven by induction on j, from m down to 1. Note first that if an active process p is not in *snap*, then the snapshot taken by p in α is a superset of *snap*. It follows that every active process with input name s_m or bigger observed s_m in its snapshot. Hence, every process entering column s_m, including s_m, descended through column s_m. Therefore, no switch is marked in column s_m, and s_m's index exiting column s_m is 1.

Given the claim for s_m through s_{j+1}, we next prove the claim for the column with index s_j. By induction the indices exiting column s_{j+1} of processes s_m through s_{j+1} are at most $2(s_m - s_{j+1}) + 1$. By part 3 of Lemma 3, their indices exiting column s_j are less than $2(s_m - s_{j+1}) + 1 + 2(s_{j+1} - s_j)$ $= 2(s_m - s_j) + 1$.

As in the argument above, only processes s_m through s_{j+1} mark any switches in column s_j. Moreover, as above these processes leave column s_j+1 with index at most $2(s_m-(s_j+1))+1 = 2(s_m-s_j)-1$. Hence no switches in column s_j are marked above this row.

This in turn implies that process s_j has index at most $2(s_m-s_j)$ exiting column s_j. The lemma follows. ∎

Given that processes have estimated entry points sufficiently high in each column, we can now establish the condition between switches set to up and those that are marked, illustrated in Figure 3.

Lemma 6 *Let α be an execution, and let c be a column index. If $R[c][i].up$ and $R[c][j].mark$ then $i \geq j$.*

Proof: Suppose $R[c][i].up$, $R[c][j].mark$ and $i < j$. Since the *mark* bits in a column are set by each processor from the bottom up, it follows that the *mark* bits below j are also set at this point, particularly $R[c][i].mark$ and $R[c][i+1].mark$. By Lemma 5, any process entering column c from the top started above entry i, setting the *up* bits from its entry point downwards. Hence, $R[c][i].up$ also implies $R[c][i+1].up$. Moreover, there exists a process p, such that p wrote $R[c][i].mark :=$ true and $R[c][i+1].mark :=$ true, and there exists a process q, such that q wrote $R[c][i+1].up :=$ true and $R[c][i].up :=$ true. Now, p set $R[c][i].mark$ and then read $R[c][i+1].up =$ false, so this write by p preceded q's setting of $R[c][i+1].up$. But similarly, q set $R[c][i+1].up$ before reading $R[c][i].mark =$ false, so q's write to $R[c][i+1].up$ preceded p's write to $R[c][i].mark$, a contradiction. The lemma follows. ∎

4.2.3 New names are distinct

The next three lemmas establish that processes compute distinct names–that the algorithm is indeed a renaming algorithm. The first of these lemmas shows that processes moving from left to right in distinct rows never collide. The second, Lemma 8, shows that each process begins moving from left to right in a distinct row. Lemma 9 combines these two observations in a simple induction to establish the renaming property.

Lemma 7 *Let α be a complete execution, and let c be a column index. If two processes compute different indices in column $c+1$ then they exit column c with distinct indices.*

Proof: If c is not the input name of any process that took steps in α, then no *up* bit is set in column c, every process retained its index from column $c+1$, and the result follows by induction. So assume that process c performed a rename operation in α.

Suppose that two processes p_j and p_{j-2} entered column c from column $c+1$ with indices i and $i-2$, and that both exited with index i from column c. (Because each process entering from a previous column either kept its index or adds two to it, this is the only possible conflict between two such processes.) Then process p_j read $R[c][i].up =$ false, and p_{j-2} read $R[c][i-2].up =$ true when computing their indices in column c.

There are two cases, depending on whether p_j saw c in its snapshot operation. If so, then p_j entered column c from the top. Since p_j appeared in its own snapshot, p_j entered column c at or above $R[c][j]$ (Lemmas 4 and 5), and since p_j later read $R[c][j].up =$ false, p_j read $R[c][j'].mark =$ true for some $j' \geq j$. But this and $R[c][j-2].up =$ true contradict Lemma 6.

So assume that p_j did not see c in its snapshot operation. Then p_j entered column c from the bottom, moving from $R[c][1]$ to $R[c][j]$, finding each *up* bit false (since it later reads $R[c][j].up =$

false), and setting every *mark* bit to true in each switch through at least $R[c][j-1]$. But again this and $R[c][j-2].up$ = true contradict Lemma 6.

It follows that processes entering column c from column $c+1$ with distinct indices exit column c with distinct indices. ■

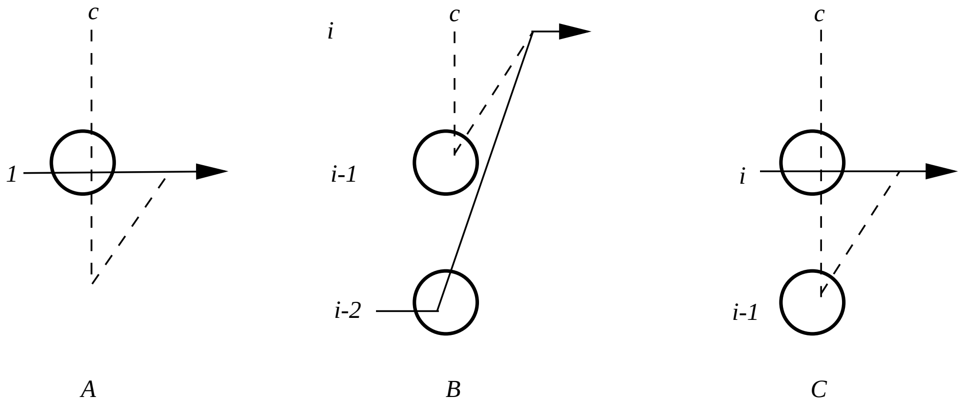

Figure 5: Cases in the proof of Lemma 8

Lemma 8 *Let α be a complete execution, and let c be a column index. If the process with input name c executed a renaming operation, it exited column c with an index that is distinct from that of any other process exiting column c.*

Proof: There are several cases to consider. First, c may have found all *mark* bits it examined to be false and computes the index 1. For this case, we consider possible conflicts with processes exiting column $c+1$ with index 1. (See Figure 5.A.) Second, c may have found $R[c][i-1].mark$ = true for some i and exited with index i. We consider possible conflicts with processes entering from column $c+1$ with index $i-2$ and i. (Figure 5.B and C.)

- (See Figure 5.A.) Process c computed index 1 after setting $R[c][1].up$:= true and finding $R[c][1].mark$ = false. Suppose another process p entered c with index 1. If p read $R[c][1].up$ = true then it computed index 3, so suppose it read $R[c][1].up$ = false. If p entered column c from the bottom, it set $R[c][1].mark$:= true and read $R[c][1].up$ = false. We have c's setting $R[c][1].up$:= true before reading $R[c][1].mark$ = false, which in turn precedes p's $R[c][1].mark$:= true which precedes reading $R[c][1].up$ = false. But this last read by p in turn precedes c's assignment $R[c][1].up$:= true, a contradiction. If p entered column c from the top, it must not have set $R[c][1].up$:= true, but read $R[c][j].mark$ = true for some $j \geq 1$. By Lemma 6, $j = 1$, and we have the same contradiction.

- Suppose then that c computes index $i > 1$, by observing $R[c][i-1].mark$ = true. If $i > 2$, by Lemma 6 $R[c][i-2].up$ = false after α, so a process entering with index $i-2$ will not compute index i. (Figure 5.B.)

  The only other possible conflict, for $i > 1$, is with a process p entering from column $c+1$ with index i, and finding $R[c][i].up$ = false when computing its index. (Figure 5.C.) Let $snap_p$ = $\{s_m, ..., s_1\}$ be the snapshot taken by process p, where $s_m > ... > s_1$.

  If p did not observe c in $snap_p$, it entered column c from the bottom, setting *mark* bits until it reached switch i or observed an *up* bit set to true. By Lemma 5, every process (including c)

that entered column c from the top entered at or above switch i. So if p observed $R[c][j].up$ = true for $j < i$, then at that point $R[c][i].up$ = true. But p later read this as false when computing its index, a contradiction. Hence, p read every up bit false up through switch i, and in particular set $R[c][i].mark$ to true before reading $R[c][i].up$ = false. But c set $R[c][i].up$ = true before reading $R[c][i].mark$ = false, leading to the usual precedence cycle.

It follows that p did observe c in $snap_p$. Since c observed $R[c][i-1].mark$ = true, some process q such that $q > c$ left column $c+1$ with index at least $i-1$ and entered column c from the bottom, setting the $mark$ bits as high as switch $i-1$. Let $snap_q = \{r_{m'}, ..., r_1\}$ be the snapshot taken by process q, where $r_{m'} > ... > r_1$. Since c is in $snap_p$ and not $snap_q$, $q \in snap_q \subset snap_p$. Let s_j be the smallest element of $snap_p$ that is bigger than c. By Lemma 5, p's index leaving column s_j is at most $2(s_m - s_j) + 1$, and by part 3 of Lemma 3, p's index, i, leaving column $c+1$ is at most $2(s_m - s_j) + 1 + 2(s_j - (c+1)) = 2(s_m - c) - 1$. But this is exactly the switch at which p entered column c. Since p later read $R[c][i].up$ = false, it must not have set $R[c][i].up$ to true. Hence it read $R[c][j].mark$ = true for some $j \geq i$, and at the time of that read, $R[c][i].mark$ = true. In turn, this means some process with incoming index i or higher entered column c from the bottom, and again by Lemma 5, c entered column c at or above switch i. Since it later read $R[c][i].mark$ = false, c must have found all $mark$ bits down to i set to false, and in particular set $R[c][i].up$ = true.

We have again that c set $R[c][i].up$ = true and later read $R[c][i].mark$ = false, and $R[c][i].mark$ was true before p read $R[c][i].up$ = false. But the later read by p preceded the assignment by c, and the read by c preceded the assignment to $R[c][i].mark$, a contradiction. It follows that p must observe $R[c][i].up$ = true, and compute $i+2$ as its index leaving column c. The lemma follows.

Lemma 9 *Let α be a complete execution, and let c be a column index. Then all processes with input names c or greater which invoke renaming operations exit column c with distinct indices.*

Proof: The proof is by induction on columns, from column $f(k)$ down to 1. Since input names are distinct, at most one process (the one with input name $f(k)$) exited column $f(k)$. So assume that all processes with input names $c+1$ or greater exited column $c+1$ with distinct indices. We must show this property holds also for column c. By Lemma 7, all the processes with input names greater than c exited column c with distinct indices. By Lemma 8, if the process with input name c invoked a renaming operation, it exited column c with an index that is distinct from that of any other process exiting column c. The lemma follows. ∎

4.2.4 Summing up

Theorem 1 *The algorithm in Figure 2 is a $(2k-1)$-renaming algorithm with $O(f(k)^2)$ step complexity.*

Proof: Lemma 4 implies the biggest name assigned is $2k-1$, and Lemma 9 implies all names assigned are distinct.

To see the time bound of $O(f(k)^2)$, consider that each process enters the array at column $c \leq f(k)$, at height $s \leq 2f(k)$, then passes through each lower-numbered column, either from a height at most $2f(k)$ down, or from the bottom to a height at most $2f(k)-1$, taking a constant number of steps at each switch, for a total of $O(f(k)^2)$ steps in the switch array. The *write* and *snapshot* operations can be carried out efficiently, e.g. in $O(k \log k)$ time, using either the Attiya and Fouren lattice-agreement algorithm or the algorithm presented in Section 2. Since $f(k) \geq k$, the result follows. ■

Corollary 1 *There is a* $(2k-1)$*-renaming algorithm with* $O(k^2)$ *step complexity.*

Proof: Processes first execute the $(6k-1)$-renaming algorithm of Attiya and Fouren [AF98], then execute the algorithm above with $f(k) = 6k - 1$. ■

5 Discussion and open problems

In this section we discuss in some detail the relationship of our $O(k^2)$, $(2k-1)$-renaming algorithm to the $O(N)$ algorithm of Attiya and Fouren, and close with a discussion of open problems and other related work.

5.1 The $O(N)$, $(2k-1)$-renaming algorithm of Attiya and Fouren [AF98].

Our $O(k^2)$ $(2k-1)$-renaming algorithm is an adaptation of the $O(N)$ algorithm of Attiya and Fouren [AF98]. As with our algorithm, the latter algorithm is based on a two-dimensional array of *reflectors*, each consisting of two binary registers. (The difference between reflectors and switches is very slight–processes accessing a reflector write one bit before reading the other, while in the switches used in our algorithm, processes read first and only write if the bit read is not set.)

In the Attiya-Fouren algorithm, each process moves vertically from the top through its input column until colliding with a horizontally-moving process or hitting the bottom of the array, and then turns and moves horizontally to the right. The row at which it exits is its output name. This array is of height $2N-1$ and width N, so each process accesses $O(N)$ reflectors, up to $2N-1$ within a single column and one in each up to N columns as it moves horizontally.

We observed first that the width of this array could be sized by setting $N = f(n)$, and that by labeling the columns of this array from right to left, so that low-numbered processes enter columns near the right edge of the array, the array is only accessed in the rightmost $f(k)$ columns. Hence, each process will only move horizontally through $f(k)$ columns. This observations changes the time complexity from $O(N)$ to $O(N + f(k))$.

The remaining problem is to change this time complexity from $O(N + f(k))$ to $O(f(k)^2)$. As was explained in Section 4.1, this is achieved by using a snapshot primitive and restricting the processes to be active in the bottom $2f(k)$ rows of the array, at the cost of up to $2f(k)$ steps in *each* column by *each* process. Hence, instead of accessing n reflectors in its entering column and one in each of as many as $f(k)$ other columns, each process may access as many as $2f(k)$ reflectors in each of $f(k)$ columns.

5.2 Open problems

The work of Attiya and Fouren [AF98], (itself building most notably on that of Moir and Anderson [MA95]) provides an interesting and powerful set of adaptive algorithms and techniques which can be used modularly to construct adaptive solutions to other fault-tolerant distributed computing problems in the read-write register model. As illustrated in Table 1, our work extends this toolkit to demonstrate novel algorithms for renaming and snapshots.

5.2.1 ϵ-agreement: constant-time algorithms

An interesting adaptive class of algorithms are those that can be solved in constant time using read/write shared memory. For example, a single column of e switches can be used to implement a simple, constant-time ϵ-agreement alglorithm, for $\epsilon = 1/e$: the bottom of the column is identified with the value 0, the top with 1, and processes enter from the top or bottom, depending upon their input value, moving as in both $(2k-1)$-renaming algorithms toward the other end of the column. The observed position i of the pivot point (or the end point of the column) is used to choose the output value i/e. Processes may differ by one value in their observation of the pivot point–resulting in a disagreement of at most $1/e$.

This raises the question: What other problems have constant complexity adaptive implementations?

5.2.2 Long-lived adaptive renaming and snapshot

In this paper we have presented mostly single-shot adaptive algorithms. Specifically, the parameter k in the complexity bounds is the total number of processes that take steps during the algorithm execution. Recently, there have been several algorithms presented for *long-lived* adaptive renaming [AAFST99, AAFST99a], and collect and snapshot [AST99], in which the parameter k is a more dynamic measure of the *actual concurrency* encountered by an operation. That is, the complexity of an operation in these implementations is bounded by a function of the maximum number of processes executing simultaneously at any point during the operation's interval.[2] Under this stronger definition of adaptiveness, the algorithms presented solve (k^2)-renaming in $O(k^2)$ step complexity [AAFST99], or $(2k-1)$-renaming, but with step complexity exponential in k [AAFST99a].

Generalizing the transformation described in Section 2, a discussion of the application of long-lived adaptive collect and snapshots to the design of other long-lived shared memory algorithms appears in [AST99].

References

[AADGMS] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., and Shavit, N. Atomic snapshots of shared memory, *Journal of the ACM*, 40(4):873-890, 1993.

[AAFST99] Afek, Y., Attiya, H., Fouren, A., Stupp, G., and Touitou, D. Long-lived renaming made adaptive, in *Proc. 18th Annual ACM Symp. on Principles of Distributed Computing*, 91-103, May 1999.

[AAFST99a] Afek, Y., Attiya, H., Fouren, A., Stupp, G., and Touitou, D. Bounded space, point-contention adaptive, long-lived renaming, Manuscript. March 1999.

[ADT95] Afek, Y., Dauber, D. and Touitou, D. Wait-free made fast, in *Proc. 23rd Annual ACM Symp. on the Theory of Computing*, 538-547, May 1995.

[AM99] Afek, Y., and Merritt, M. Adaptive, wait-free $(2k-1)$-renaming, in *Proc. 18th Annual ACM Symp. on Principles of Distributed Computing*, 105-112, May 1999.

[2] A weaker definition of the parameter k is also studied, in which the complexity of an operation is bounded by a function of the total number of other processes active during any part of the operation interval.

[AH90] Aspnes, J. and Herlihy, M. Fast randomized consensus using shared memory, *Journal of Algorithms*, 441-461, September 1990.

[ABND+90] Attiya, H., Bar-Noy, A., Dolev, D., Koller, D., Peleg, D., and Reischuk, R. Renaming in an asynchronous environment, *Journal of the ACM* 37(3)524-548, July 1990.

[AF98] Attiya, H. and Fouren, A. Adaptive wait-free algorithms for lattice agreement and renaming, in *Proc. 17th Annual ACM Symp. on Principles of Distributed Computing*, 277-286, 1998.

[AF98a] Attiya, H. and Fouren, A. Adaptive wait-free algorithms for lattice agreement and renaming, Technion Computer Science Department Technical Report #0931, April 1998.

[F99] Fouren, A. Exponential examples for two renaming operations (Draft), http://www.cs.technion.ac.il/~hagit/pubs/expo.ps.gz, August, 1999.

[AR93] Attiya, H. and Rachman, O. Atomic snapshots in $O(n \log n)$ operations, in *Proc. 12th Annual ACM Symp. on Principles of Distributed Computing*, 29-40, 1993.

[AST99] Afek, Y., Stupp, G., and Touitou, D. Long-lived and adaptive collect with applications, in *Proc. of 40th IEEE Symp. on Foundations of Computer Science*, Oct. 1999.

[BD89] Bar-Noy, A. and Dolev, D. Shared memory versus message-passing in an asynchronous distributed environment, In *Proc. 8th Annual ACM Symp. on Principles of Distributed Computing*, 307-318, 1989.

[BG93] Borowsky, E. and Gafni, E. Immediate atomic snapshots and fast renaming, in *Proc. of the 12th Annual ACM Symp. on Principles of Distributed Computing*, pages 41–51, 1993.

[BP89] Burns, J. and Peterson, L. The ambiguity of choosing, in *Proc. 8th Annual ACM Symp. on Principles of Distributed Computing*, 41-51, 1989.

[G92] Gafni, E., More about renaming: Fast algorithm and reduction to the k-set test-and-set problem, unpublished manuscript, 1992.

[GLS95] Gawlick, R., Lynch, N. and Shavit, N. Concurrent timestamping made simple, in *Proc. of the 1st Israel Symp. on the Theory of Computing and Systems*, Springer-Verlag, 171-185, May 1992.

[HS93] Herlihy, M. and Shavit, N. The asynchronous computability theorem for t-resilient tasks, in *Proc. 25th ACM Symp. on the Theory of Computing*, 111–120, May 1993.

[Lam78] Lamport, L. Time, clocks and the ordering of events in a distributed system, *Communications of the ACM*, 21(7):558-565, July 1978.

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

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