Distributed Computing manuscript No.
(will be inserted by the editor)

An Adaptive Collect Algorithm with Applications

Hagit Attiya[⋆1], **Arie Fouren**[1], **Eli Gafni**[2]

[1] Department of Computer Science, The Technion, Haifa 32000, Israel. hagit@cs.technion.ac.il, leonf@cs.technion.ac.il.
[2] Computer Science Department, UCLA. eli@cs.ucla.edu.

The date of receipt and acceptance will be inserted by the editor

Summary. In a shared-memory distributed system, n independent asynchronous processes communicate by reading and writing to shared variables. An algorithm is *adaptive* (to total contention) if its step complexity depends only on the actual number, k, of active processes in the execution; this number is unknown in advance and may change in different executions of the algorithm. Adaptive algorithms are inherently *wait-free*, providing fault-tolerance in the presence of an arbitrary number of crash failures and different processes' speed.

A wait-free adaptive collect algorithm with $O(k)$ step complexity is presented, together with its applications in wait-free adaptive algorithms for atomic snapshots, immediate snapshots and renaming.

Key words: asynchronous shared-memory systems – contention-sensitive complexity – wait-free algorithms – read/write registers – atomic snapshots – immediate snapshots – renaming

1 Introduction

An *asynchronous shared-memory system* consists of n asynchronous processes, each with a distinct identifier, communicating by reading and writing to shared variables. *Wait-free* algorithms [25] guarantee that a process completes its operation within a finite number of its own steps regardless of the behavior of other processes.

In a wait-free algorithm, processes typically collect up-to-date information from each other by reading from an array indexed with process' identifiers. Since distributed algorithms are designed to accommodate a large number of processes, this scheme is an over-kill when few processes participate in the algorithm: many entries are read although they contain irrelevant information about processes not wishing to coordinate. An *adaptive* algorithm alleviates this concern as its step complexity expression is bounded by a function of the number of processes that participate in the algorithm (the *active* processes).

This paper presents an algorithm for collecting up-to-date information whose step complexity adjusts to the number of active processes. We also present several applications of this algorithm, demonstrating a modular way to obtain adaptive algorithms for other problems.

Our adaptive wait-free collect algorithm (presented in Section 3) has $O(k)$ step complexity, where k is the number of active processes. Clearly, any algorithm that requires $f(k)$ stores and collects (for some function f) can be made adaptive by substituting our collect algorithm. More sophisticated usage of the collect algorithm is required in order to obtain adaptive wait-free algorithms for atomic snapshots and immediate snapshots. Adaptive atomic snapshots and immediate snapshots, in turn, imply adaptive renaming algorithms.

Atomic snapshots [1] provide instantaneous global views of the shared memory; they are widely accepted as a tool for simplifying the design of wait-free algorithms. Our atomic snapshots algorithm (Section 4) is based on [15] and it has $O(k \log k)$ step complexity.

Immediate snapshots [19] extend atomic snapshots, and guarantee that no process obtains a view that is *strictly* between an update of process p_j and the following view p_j obtains; they were used for renaming [19] and to study wait-free solvable tasks [16,18,29]. Our immediate snapshots algorithm (Section 5) is based on [7,17] and it has $O(k^3)$ step complexity.

In the M*-renaming* problem [10], each process starts with a distinct name in some range and is required to choose a distinct name in a smaller range of size M. In the more general *long-lived* M-renaming problem [28], processes repeatedly *acquire* and *release* names. Adaptive versions of well-known wait-free $(2k-1)$-renaming algorithms are easily obtained with adaptive atomic snapshots and immediate snapshots (see [24]). This includes a one-shot algorithm with $O(k^3)$ step complexity [19], which is presented in Section 6.

In another paper [13], we present efficient adaptive wait-free algorithms for lattice agreement (one-shot atomic snapshots) and $(6k-1)$-renaming; these algorithms do

⋆ Work supported by the fund for the promotion of research in the Technion.

not use a collect procedure and their step complexity is $O(k \log k)$. Afek and Merritt [4] use them to obtain an adaptive wait-free $(2k-1)$-renaming algorithm, with $O(k^2)$ step complexity.

Several papers [9,27,28] study algorithms whose step complexity depends only on n, and not on the range of process' identifiers. These algorithms provide a weaker guarantee than adaptive algorithms, whose step complexity adjusts to the actual number of active processes, which can be much lower than the upper bound, n. Anderson and Moir [9] present an adaptive renaming algorithm that uses the (stronger) test&set memory access operation.

The algorithms presented in this paper adapt to the *total* contention—if a process ever performs a step, then it influences the step complexity of the algorithm throughout the execution. More useful are algorithms that adapt to the *current* contention, that is, whose step complexity depends only on the number of currently active processes. Our collect algorithm is a building block in a long-lived renaming algorithm [2,12], whose step complexity adapts to the current contention. The long-lived renaming algorithm, in turn, is used in a collect algorithm [5] with $O(k^3)$ step complexity, where k is the current contention. This collect algorithm is used to extend our immediate snapshot algorithm to be long-lived and adapt to current contention [6] (with $O(k^4)$ step complexity). Afek, Dauber and Touitou [3] introduce implementations of long-lived objects whose step complexity is linear in the current contention; however, they use strong load-linked and store-conditional operations.

Lamport [26] suggests a mutual exclusion algorithm that requires a constant number of steps when a single process wishes to enter the critical section, using reads and writes; when several processes compete for the critical section, the complexity depends on the range of names. Choy and Singh [21] present mutual exclusion algorithms, using reads and writes, which are adaptive in an amortized sense; in the worst case, the step complexity of their algorithms depends on n. (Alur and Taubenfeld [8] show that this is inherent.)

2 Preliminaries

We consider n *processes*, $p_1, \ldots, p_n$; each process p_i is modeled as a (possibly infinite) state machine, with a unique name $id_i \in \{0, \ldots, N-1\}$. Processes communicate by read and write operations on shared variables; a read(R) operation does not change the state of R and returns the current state of R; a write(v, R) operation changes the state of R to v. Registers are *multi-writer multi-reader*, allowing read and write operations by all processes.

An *event* is a computation step by a single process; in an event, a process determines the operation to perform according to its local state, and determines its next local state according to the value returned by the operation.

An *execution* α is a (finite or infinite) sequence of events $\phi_0, \phi_1, \phi_2, \ldots$. For every $r = 0, 1, \ldots$, if p_i is the process performing the event ϕ_r, then it applies a read or a write operation to a single register and changes its state according to its transition function. There are no constraints on the interleaving of events by different processes, reflecting the assumption that processes are *asynchronous* and there is no bound on their relative speeds.

A process is *active* in an execution α if it takes a step in α. Let $k(\alpha)$ be the number of active processes in α.

An algorithm specifies procedures to be invoked when a process performs an operation. The *interval* of an operation op_i by process p_i is the execution segment between the first event and the last event of p_i in op_i. If the last event of p_i in op_i is before the first event of process p_j in an operation op_j, then op_i *precedes* op_j and op_j *follows* op_i.

For an execution segment β, let $step(\beta, p_i)$ be the number of read/write operations performed by process p_i in β.

Algorithm A is *adaptive (to total contention)* if there is a function $f : \mathcal{N} \mapsto \mathcal{N}$ such that the following holds for every execution α of A: if process p_i has an operation interval β in α, then $step(\beta, p_i) \leq f(k(\alpha))$. Namely, the step complexity of an operation depends only on the number of active processes in α.

A *wait-free* algorithm guarantees that every process completes its computation in a finite number of steps, regardless of the behavior of other processes. Since $k(\alpha)$ is bounded (it is at most n), $f(k(\alpha))$ is also bounded; hence, an adaptive algorithm must be wait-free.

A *view* V is a set of process-value pairs, $\{\langle p_{i_1}, v_{i_1}\rangle, \ldots\}$, without repetitions of processes. $V(id_j)$ refers to v_j, if $\langle p_j, v_j\rangle \in V$, and to $\perp$ otherwise.

A solution for the collect problem provides algorithms for two operations—store and collect. A store(val) operation of p_i declares val as the latest value for p_i, and a collect operation returns the latest values stored by active processes. Formally, a collect operation cop returns a view V such that the following holds for every process p_j: if $V(p_j) = \perp$, then no store operation of p_j precedes cop; if $V(p_j) = v \neq \perp$ then v is the value of a store operation sop of p_j that does not follow cop, and there is no other store operation sop' of p_j that follows sop and precedes cop. That is, cop does not read from the future or miss a preceding store operation.

Moreover, if a collect operation op follows another collect operation cop', then cop should return a view that is more up-to-date. To capture this notion, we define a partial order on views: $V_1 \preceq V_2$, if for every process p_i such that $\langle p_i, v_i^1\rangle \in V_1$, we have $\langle p_i, v_i^2\rangle \in V_2$, and v_i^2 is written in a store operation of p_i that follows or is equal to a store operation of p_i that writes v_i^1. We require that if cop precedes cop', then $V_1 \preceq V_2$.

The collect problem is easily solved using an array indexed with processes' names: A process stores its most recent value to the entry indexed with its name; to collect, a process reads the entire array. (This can be used as an alternative definition, cf. [5].) In this scheme, a collect requires $O(N)$ steps.

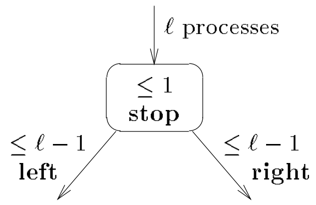


Fig. 1. A splitter.

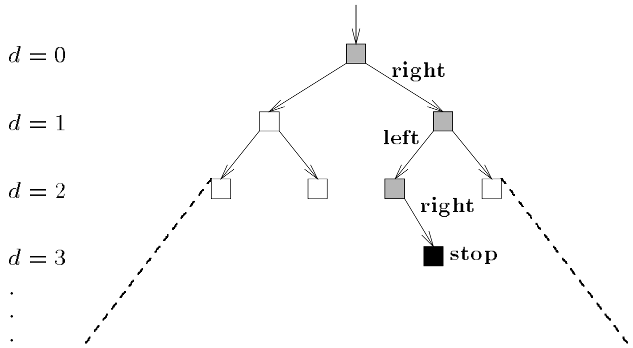


Fig. 2. An execution of register in a complete binary tree of splitters.

3 Adaptive Collect

We present an adaptive wait-free algorithm for store and collect, with $O(k)$ step complexity. The algorithm uses the *splitter* suggested by Moir and Anderson [28]: A process entering a splitter exits with either **stop**, **left** or **right**. It is guaranteed that if a single process enters the splitter, then it obtains **stop**, and if two or more processes enter the splitter, then there are two processes that obtain different values. (See Figure 1) Thus the set of processes is "split" into smaller subsets, according to the values obtained.

The collect algorithm uses a complete binary tree of depth $n - 1$, with splitters in the vertices. In its first store, a process acquires a vertex v; from this point on, the process stores its up-to-date values in $v.val$.

A process acquires a vertex in the tree using procedure register; in register, the process starts at the root and moves down the tree according to the values obtained in the splitters along the path: If it receives **left**, it moves to the left child; if it receives **right**, it moves to the right child. A process marks each vertex it accesses by raising a flag associated with the vertex; a vertex is *marked*, if its flag is raised. The process acquires a vertex v when it obtains **stop** at the splitter associated with v; then it writes its id into $v.id$. (See Figure 2.)

To perform a collect, a process traverses the part of the tree containing marked vertices, in DFS order, and collects the values written in the marked vertices.

A simple implementation of a splitter [28] is based on Lamport's mutual exclusion algorithm [26], and uses two shared variables, X and Y. Initially, $X = \perp$ and $Y =$ **false**. A process executing the splitter first writes its id into X and then reads Y. If $Y =$ **true**, then the process returns **right**. Otherwise, the process sets $Y =$ **true** and checks X. If X still contains its id, then the process returns **stop**; if X does not contain its id, then the process returns **left**. The following lemma, from [28], states the main properties of the splitter.

Lemma 1. *If ℓ processes access a specific splitter, then the following conditions hold:*
(1) at most one process obtains **stop** *in this splitter,*
(2) at most $\ell - 1$ processes obtain **left** *in this splitter, and*
(3) at most $\ell - 1$ processes obtain **right** *in this splitter.*

The code for collect and store, as well as for the splitter, appears in Algorithm 1. In the algorithm, the following shared variables are associated with each vertex v in the tree:

mark:	Indicates whether some process accessed v; initially **false**.
id:	Holds the identifier of the process that stops in v; initially $\perp$.
value:	Holds an updated value of the process that stops in v; initially $\perp$.
X:	Holds a process' identifier, for the splitter associated with v; initially $\perp$.
Y:	Holds a Boolean value, for the splitter associated with v; initially **false**.
left-child:	Pointer to the left child of v.
right-child:	Pointer to the right child of v.

To prove the correctness and complexity of the algorithm, fix an execution α of the algorithm and let k be the number of processes that call store at least once in α.

Lemma 2. *If the depth of a vertex v is d, $0 \leq d \leq k$, then at most $k - d$ processes access v.*

Proof. The proof is by induction on d, the depth of v. In the base case, $d = 0$, the lemma trivially holds since at most k processes are active.

For the induction step, suppose that the lemma holds for vertices at depth d, $0 \leq d < k$, and consider some vertex v with depth $d+1$. Let u be v's parent in the tree. The depth of u is d, and by the inductive hypothesis, at most $k - d$ processes access u. If v is the left child of u, then Property (2) of the splitter (Lemma 1) implies that at most $k - d - 1$ of the processes obtain **left** at u and access v. If v is the right child of u, then Property (3) of the splitter (Lemma 1) implies that at most $k - d - 1$ of the processes obtain **right** at u and access v. □

By Lemma 2 and the algorithm, when a process performs register, it stops in a vertex with depth less than or equal to $k-1$. By Property (1) of the splitter (Lemma 1), at most one process stops in each vertex. Therefore, we have the following lemma:

Lemma 3. *Each process writes its id in a vertex with depth $\leq k - 1$ and no other process writes its id in the same vertex.*

Since each splitter requires a constant number of operations, Lemma 3 implies that the step complexity of

Alg. 1 store and collect: code for process p_i.

```
shared variables:
    CollectTree : complete binary tree of splitters
        of depth n − 1
local variables:      // persistent across invocations of store
    descriptor : vertex, initially ⊥

void procedure store(val)                        // update value
    if ( descriptor == ⊥ ) then                    // first time
        descriptor = register()
    descriptor.value = val

vertex procedure register()                    // acquire a vertex
    v = CollectTree.root
    repeat
        v.mark = true
        move = splitter(v)  // returns stop, left, or right
        if ( move == left ) then v = v.left-child
        if ( move == right ) then v = v.right-child
    until ( move == stop )
    v.id = id_i                         // write your identifier
    return v                             // location descriptor

view procedure collect()
            // collect updated values of active processes
    return( DFS(∅, CollectTree.root) )

view procedure DFS(V : view; v : vertex)
        // DFS traversal of the marked part of the tree
    if ( v.mark ) then
        if ( v.value ≠ ⊥ ) then V = V ∪ {⟨v.id, v.value⟩}
        V = V ∪ DFS(V, v.left-child)
        V = V ∪ DFS(V, v.right-child)
    return(V)

{left, right, stop} procedure splitter(v : vertex)
                            // from Moir and Anderson [28]
    v.X = id_i                          // write your identifier
    if ( v.Y ) then return(right)
    v.Y = true
    if ( v.X == id_i ) then return(stop) // check identifier
    else return(left)
```

register is $O(k)$. This implies that the first invocation of store requires $O(k)$ steps; clearly, all later invocations of store require only $O(1)$ steps. In addition, this lemma implies that the location descriptors returned by register are unique.

If a vertex v is marked, then some process p_i sets $v.mark$ to **true**; this process also marks all the vertices on the path from the root to v before marking v. Since no process resets *mark* variables to **false**, this implies the next lemma:

Lemma 4. *All vertices on the path from the root to a marked vertex v are marked.*

Assume that *cop* is a collect of process p_i. If some process p_j completes its first store before p_i starts *cop*, then p_j writes id_j into $v.id$, for some vertex v, before p_i starts *cop*. By Lemma 4, all vertices on the path from the root to v are marked, and therefore, p_i visits v during the DFS traversal in *cop*. The algorithm implies that p_i reads p_j's most up-to-date value from $v.value$. Clearly, p_i cannot read a value written by a store of p_j that follows *cop*. Moreover, since values are updated with a single operation, a later collect returns a more up-to-date view. This implies that Algorithm 1 solves the collect problem.

Theorem 1. *Assume a* collect *operation cop returns a view V. Then the following holds for every process p_j:*
(1) if $V(p_j) = \perp$, then no store *operation of p_j precedes cop;*
(2) if $V(p_j) = v \neq \perp$, then v is the value of a store *operation sop of p_j that does not follow cop, and there is no other* store *operation sop' of p_j that follows sop and precedes cop;*
(3) if cop precedes a collect *operation cop' that returns a view V', then $V \preceq V'$.*

The step complexity of collect is linear in the number of vertices in the marked tree that are traversed by procedure DFS. We prove that this number is at most $2k-1$. (This holds despite the fact that the depth of the marked tree can be k, which in general implies only a bound of 2^k on the number of marked vertices.)

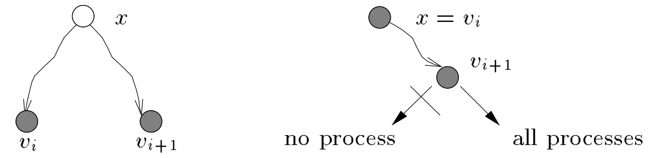
Consider a collect operation, and let α be the shortest finite execution prefix that contains its execution interval. Let $S = v_0, v_1, \ldots, v_l$ be the vertices of the marked tree after α, appearing in an in-order; i.e., for every marked vertex, v, the vertices of the left sub-tree of v appear before v in S and the vertices of the right subtree of v appear after v in S.

A vertex $v \in S$ is *grey* if there is a process that accesses the left child of v in α (that is, writes **true** in the variable *mark* of *v.left-child*) and there is a process that accesses the right child of v in α (that is, writes **true** in the variable *mark* of *v.right-child*). A marked vertex that is not grey is *black*. For any black vertex v, *(a)* there is a process that sets $v.mark$ to **true**, and *(b)* one of the children of v (e.g., *v.right-child*) is not accessed by any process in α. By Lemma 1(2), not all the processes accessing v return **left**. Therefore, at least one process that accesses v either returns **stop** in v, or fault-stops in v.

Lemma 5. *There is a black vertex between every pair of grey vertices in S.*

Proof. Suppose, by way of contradiction, that there are two consecutive grey vertices v_i and v_{i+1} in S. We first prove that one of them is an ancestor of the other in the marked tree. Let x be the lowest common ancestor of v_i and v_{i+1} in the marked tree; by Lemma 4, x is marked. If $x \neq v_i$ and $x \neq v_{i+1}$, then v_i belongs to the left subtree of x, and v_{i+1} belongs to the right subtree of x (see Figure 3(a)). Since S is an in-order traversal of the marked subtree, x appears between v_i and v_{i+1} in S, contradicting the assumption that v_i and v_{i+1} are consecutive in S.

Suppose that v_i is an ancestor of v_{i+1}. Since v_i appears before v_{i+1} in the in-order sequence S, v_{i+1} belongs to the right subtree of v_i. Since v_{i+1} appears immediately after v_i in S, the left child of v_{i+1} is unmarked, contradicting the fact that v_{i+1} is grey. (See Figure 3(b).)



(a) (b)

Fig. 3. Illustrations for the proof of Lemma 5.

A similar argument can be applied if v_{i+1} is an ancestor of v_i. Since v_i appears before v_{i+1} in the in-order sequence S, v_i belongs to the left subtree of v_{i+1}. Since v_i appears immediately before v_{i+1} in S, the right child of v_i is unmarked, contradicting the fact that v_i is grey. □

If the first vertex in S, v_0, is grey, then the left subtree of v_0 must contain marked vertices. Therefore, some vertex precedes v_0 in S, which is a contradiction. A similar argument shows that the last vertex in S is black, implying the next lemma:

Lemma 6. *The first and the last vertices in S are black.*

With each black vertex we can associate a distinct active process that accesses the vertex and does not go below it in α; thus, there are at most k black vertices. Therefore, the number of grey vertices is at most $k-1$, by Lemma 5 and Lemma 6. Hence, the marked tree contains at most $2k-1$ vertices. Thus, procedure DFS visits at most $2k-1$ vertices, each requiring a constant number of operations, implying that the step complexity of collect is $O(k)$.

Theorem 2. *Algorithm 1 solves the collect problem, with $O(k)$ step complexity.*

4 Adaptive Atomic Snapshots

The *atomic snapshot* problem [1] extends the collect problem by requiring views to look instantaneous. Instead of separate update and store operations, we provide a combined upscan operation, which updates a new value and atomically collects a view. The returned views should satisfy the following conditions (cf. [14]):

Validity: If an upscan operation *op* returns a view V, and precedes an upscan operation op', then V does not include the value written by op'.[1]

Self-inclusion: The view returned by the ℓth upscan operation of p_j includes the ℓth value written by p_j.

Comparability: If V_1 and V_2 are the views returned by two upscan operations, then either $V_1 \preceq V_2$ or $V_2 \preceq V_1$.

An efficient adaptive atomic snapshot algorithm, with $O(k \log k)$ step complexity, can be derived from the algorithm solving the problem for n processes with $O(n \log n)$ steps [15]. This transformation is not trivial since in the

[1] Typically, this condition trivially holds and we do not prove it below.

Alg. 2 The classifier procedure (from [15]): code for p_i

```
procedure classifier(M : integer; I_i: view)
        returns {left, right} and a view
   store(I_i)
   {R_1, ..., R_n} = collect
   if |∪{R_1, ..., R_n}| > M then
       {R_1, ..., R_n} = collect
       return(right, ∪{R_1, ..., R_n})
   else return(left, I_i)
```

non-adaptive algorithm, processes descend down a binary tree of depth $O(\log n)$ (see below); thus, the number of stores and collect depends on n. We describe the one-shot algorithm; it can be made long-lived using techniques of [14,15,22].

The non-adaptive algorithm uses a complete binary tree of depth $\log n$, whose vertices are labeled as a search tree, in which all values are stored in the leaves: The leaves are labeled $1, 2, \ldots$ from left to right; the label of an inner vertex is equal to the label of the right-most leaf in its left subtree (Figure 4). A simple classifier procedure (Algorithm 2) is associated with each vertex; the procedure takes a threshold value and an input view as parameters; it returns a *side* (left or right) and a view. Procedure classifier separates operations so that less knowledgeable operations proceed to the left, and more knowledgeable operations proceed to the right (see Lemma 7).

An upscan operation traverses the tree downwards from the root. In each inner vertex v, the operation calls classifier with $Label(v)$ as the threshold parameter and the view it obtained in the previous vertex (in the root, the view contains only the operation's value); the operation continues left or right, according to the side returned by classifier. The operation terminates at a leaf and returns the view obtained in the last inner vertex (without performing classifier in the leaf). The following simple lemma [15, Lemma 3.1] states the properties of classifier.

Lemma 7. *Assume* classifier *is called with threshold parameter M, and that process p_i obtains the view O_i from* classifier*. Then the following holds:*
*(1) $|\cup\{O_j \mid p_j$ returns **left**$\}| \leq M$, and*
*(2) if p_i returns **right**, then $|O_i| > M$ and O_i contains $\cup\{O_j \mid p_j$ returns **left**$\}$.*

The adaptive algorithm uses an unbalanced binary tree, constructed from $\log n$ complete binary trees of exponentially growing sizes ($1, 2, 2^2, \ldots$ leaves), connected by a single path (Figure 4). As in the non-adaptive algorithm, leaves are labeled $1, 2, \ldots$ from left to right and an inner vertex is labeled with the label of the right-most leaf in its left subtree.

First, note that views returned by two operations, op_i and op_j, returning V_i and V_j from different leaves are comparable. Let v be the minimal common ancestor of these leaves; v is an inner vertex. Clearly, op_i and op_j calls classifier at v, and one of them (say, op_i) returns **left**, while the other (say, op_j) returns **right**.

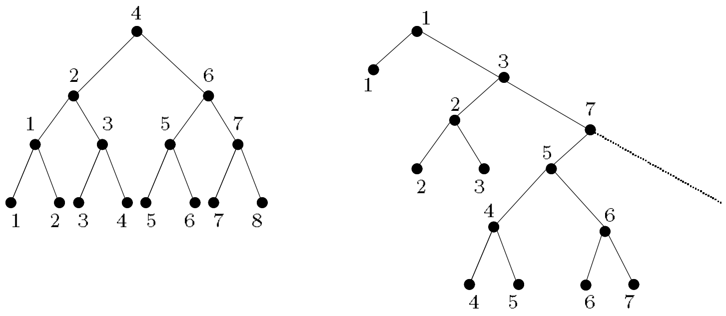


Fig. 4. Trees for lattice agreement: Non-adaptive (left) and adaptive (right).

The algorithm implies that V_i is contained in $\cup\{O_l \mid p_l$ returns **left** and O_l in $v\}$, and that V_j contains the view p_j returns from classifier in v. By Lemma 7(2), V_j contains V_i.

Next, consider two operations, op_i and op_j, returning V_i and V_j from the same leaf v; denote $M = \mathit{Label}(v)$. Let u be the last vertex (on the path from the root to v) in which op_i and op_j go right[2]; v is the left-most leaf in the right subtree of u and by construction, $\mathit{Label}(u) = M-1$. By Lemma 7(2), $|V_i|, |V_j| > M-1$. Let w be the last vertex in which op_i and op_j go left; v is the right-most leaf in the left subtree of w and hence, $\mathit{Label}(w) = \mathit{Label}(v) = M$. As mentioned above, the algorithm implies that V_i and V_j are contained in $\cup\{O_l \mid p_l$ returns **left** and O_l in $v\}$. By Lemma 7(1), this union contains at most $\mathit{Label}(w)$ values. Thus, $|V_i \cup V_j| \leq M$, so $V_i = V_j$.

This allows to prove that the snapshot algorithm is correct, along the lines of [15]. An operation accesses at most $O(\log k)$ vertices on its way to a leaf. Since classifier in each vertex requires $O(k)$ steps, the algorithm requires $O(k \log k)$ steps.

5 Adaptive Immediate Snapshots

The *immediate snapshot* problem [19] provides a combined im-upscan operation, updating a new value and returning a view. In addition to the validity, self-inclusion, and comparability properties of the atomic snapshot problem, returned views should satisfy the next condition:

Immediacy: If the view returned by some im-upscan operation, V_1, includes the value written in the ℓth im-upscan of p_j that returns the view V_2, then $V_2 \preceq V_1$.

5.1 Overview of the Algorithm

For ease of exposition, a view is represented by a set of counters, holding the number of updates performed by processes. Each process owns an unbounded array of values.[3] In an im-upscan operation, a process writes the new value into its array and increments a counter holding the number of values it has written; then it obtains a view of the counters (which can be used to retrieve the values from the arrays). The *sum* of counters in a view V, denoted $\sum V$, is the total number of updates preceding V; clearly, for views satisfying the comparability property $\sum V_1 \leq \sum V_2$ if and only if $V_1 \preceq V_2$.

The overall structure of our algorithm follows the non-adaptive immediate snapshot algorithm [7,17]. Process p_i first finds an atomic snapshot view V containing its new value. V is written in a *floor* whose number s_i is equal to $\sum V$; clearly, all views written in the same floor are equal. (The number of floors is infinite, since $\sum V$ is unbounded for a long-lived algorithm.) Then, p_i participates in a distinct copy of one-shot immediate snapshot in each floor below s_i, with its new value as input, until it sees its previous value in the view written in one of these floors. When this happens, p_i returns the maximal values from the view written in this floor and the view it obtained in the one-shot immediate snapshot of this floor.

To bound the number of floors process p_i accesses, it takes as V the *smallest* atomic snapshot view containing its new value among the snapshots obtained by other processes. That is, s_i is the smallest floor where a view containing p_i's new value is written. This is used below (Lemma 11) to show that p_i accesses at most k floors. To allow p_i to find the smallest view containing its new value, each process p_j maintains an array which holds, for every process p_l, the first view p_j observes with the most recent value of p_l; p_j updates this view whenever it sees a new value for p_l. To find V and calculate its start floor, p_i reads the appropriate entries of the active processes' arrays and picks the minimal view containing its last value.

Since the view written in floor s_i contains p_i's new value, processes returning from floors $\geq s_i$ see this (or a later) value of p_i. Since p_i returns from some floor $\mathit{ff} < s_i$ containing its previous value, processes returning from floors $< \mathit{ff}$ see previous values of p_i. Since p_i performs the one-shot immediate snapshot algorithm with its last value as input in each floor between s_i and ff, the views returned from these floors include this value. The one-shot immediate snapshot in floor ff guarantees that views returned from floor ff satisfy the immediacy property.

The one-shot immediate snapshot algorithm used in each floor [19] relies on the number of participants: processes start at level n, and descend through levels until some condition is met. We notice that processes need not start at the same level: they only have to start at (pos-

[2] If u does not exist, then $M = 1$ and a simpler version of the following argument can be applied.

[3] This array can be bounded, see [7].

sibly different) levels that are larger than the number of processes participating in the one-shot immediate snapshot algorithm (see Section 5.4). Below, we show how a process picks its start level for floor f to be larger than or equal to k_f, the number of processes participating in floor f. The step complexity of the one-shot algorithm used in each floor depends on the start level, which is smaller than or equal to $k+1$, making the algorithm adaptive.

5.2 Details of the Algorithm

Algorithm 3 uses an infinite number of floors. A copy of the adaptive one-shot immediate snapshot algorithm (Algorithm 4, presented below), denoted $\mathsf{os\text{-}im\text{-}upscan}_f$, is associated with every floor f, as well as the following data structures:

1. $view[f]$, a view; initially contains the empty view, $\perp$.
2. $flag[f][1 \ldots N]$ an array of bits, one for each process; initially, all bits are **false**.

Each process p_i maintains an array $A_i[0, \ldots, N-1]$ of views; $A_i[id_j]$ holds the first view containing the last value of p_j, among the views observed by p_i.

After obtaining a view V', p_i checks, for each process $p_j \in V'$, whether p_j incremented its counter since p_i's previous $\mathsf{im\text{-}upscan}$ operation. If it did, then p_i writes V' (containing the new counter of p_j) into $A_i[id_j]$. To find its start floor, p_i chooses the minimal view V containing its new counter, among the views stored for it by other processes. To keep the step complexity of the algorithm adaptive, p_i reads $A_j[id_i]$ only for processes $p_j \in V'$. Then p_i writes V in its start floor, whose number is $\sum V$.

Process p_i continues the algorithm one floor below its start floor. In each floor f, if p_i reads a non-$\perp$ view from $view[f]$ then p_i sets $flag[f][i]$ to **true**. Then, p_i obtains a view W_i^f from $\mathsf{os\text{-}im\text{-}upscan}_f$. If $view[f]$ does not contain the last value of p_i and the flag of one of the processes in W_i^f is **true**, then p_i returns a view containing the maximal counters from W_i^f and $view[f]$; otherwise, p_i accesses floor $f-1$.

Clearly, processes appearing in p_i's initial view, V_i, may access floors below p_i's start floor. In addition, processes may descend from higher floors. These processes "register" in the floor before participating in the one-shot immediate snapshot associated with it. To allow registration, a distinct copy of store and $\mathsf{collect}$ (Algorithm 1), denoted store_f and $\mathsf{collect}_f$, is associated with each floor f. A process registers before accessing floor f, using store_f. Process p_i collects a set U_i of processes registered in its start floor, s_i, using $\mathsf{collect}_{s_i}$. $|V_i \cup U_i|+1$ is the start level parameter of p_i for $\mathsf{os\text{-}im\text{-}upscan}$ in all floors it accesses. Since V_i and U_i contain only active processes, $|V_i \cup U_i| + 1 \leq k+1$.

Note that different invocations of $\mathsf{im\text{-}upscan}$ by process p_i do not call the same copy of $\mathsf{os\text{-}im\text{-}upscan}$. If an operation op_1 of p_i starts in floor f, then op_1 calls $\mathsf{os\text{-}im\text{-}upscan}$ only in floors $< f$, and the view written in floor f contains the value of op_1. A later operation op_2 of p_i reads this value (or a later one) from a floor $\geq f$; therefore op_2 returns from a floor $\geq f$ and does not call $\mathsf{os\text{-}im\text{-}upscan}$ in floors $< f$.

5.3 Proof of Correctness and Complexity Analysis

Our key lemma proves that only processes in $V_i \cup U_i$ may access floors $1, \ldots, s_i - 1$; that is, $start\text{-}level_i$ is larger than or equal to the number of processes in the floors p_i accesses.

Lemma 8. *If p_i starts at floor s_i, and p_j accesses a floor $f < s_i$, then $p_j \in U_i \cup V_i$.*

Proof. If $p_j \in V_i$, then the lemma clearly holds. Otherwise, the atomic snapshots properties imply that p_j accesses floor s_i, before it accesses floor f.

If p_j completes store_{s_i} before p_i starts $\mathsf{collect}_{s_i}$, then $p_j \in U_i$, and the lemma follows.

Otherwise, p_j reads $V_i \neq \perp$ from $view[s_i]$ since it reads after completing $\mathsf{store}_{s_i}(id_j)$, and p_i writes V_i into $view[s_i]$ before starting $\mathsf{collect}_{s_i}$. Since $p_j \notin V_i$, it follows that p_j evaluates the condition in Line 15 to **true**, and returns from floor s_i, which is a contradiction. □

If p_i returns V_i and p_j returns V_j from the same floor f, then they read the same value from $view[f]$. W_i^f and W_j^f are views returned by $\mathsf{os\text{-}im\text{-}upscan}_f$ and hence, they are comparable. Thus, V_i and V_j are comparable. The comparability property is proved by showing that views returned from different floors are comparable; the proof follows [17, Lemma 3.3.2].

Lemma 9. *If p_i returns V_i from floor f_i and p_j returns V_j from floor $f_j < f_i$, then $V_j \preceq V_i$.*

Proof. Since views written in the floors are ordered by containment, $view[f_j] \preceq view[f_i]$. We show that $W_j^{f_j}(p_k) \leq V_i(p_k)$, for any process p_k.

The lemma trivially holds if $W_j^{f_j}(p_k) = \perp$. Otherwise, $\langle p_k, l\rangle \in W_j^{f_j}$, for some l; thus, p_k participates in $\mathsf{os\text{-}im\text{-}upscan}_{f_j}$ (on floor f_j) during its l'th immediate snapshot, which starts at floor s_k. The lemma clearly holds if $s_k < f_i$, since $\langle p_k, l\rangle \in view[s_k] \preceq view[f_i] \preceq V_i$.

If $s_k \geq f_i$, then p_k accesses floor f_i, evaluates the condition in Line 15 to **false**, and goes to a lower floor. If p_k reads a non-$\perp$ value from $view[f_i]$ that includes $\langle p_k, l\rangle$, then $\langle p_k, l\rangle \in view[f_i] \preceq V_i$, since p_i reads the same non-$\perp$ value from $view[f_i]$.

Otherwise, p_k reads **false** from $flag[f_i][x]$, for every process $p_x \in W_k^{f_i}$. Clearly, p_i reads **true** from $flag[f_i][y]$, for some process $p_y \in W_i^{f_i}$. However, p_y writes **true** to $flag[f_i][y]$ before calling $\mathsf{os\text{-}im\text{-}upscan}$, and p_k must read **true** from $flag[f_i][y]$ if $p_y \in W_k^{f_i}$. Thus, $p_y \notin W_k^{f_i}$, and by the comparability property, $W_k^{f_i} \subset W_i^{f_i}$. The self-inclusion property of $\mathsf{os\text{-}im\text{-}upscan}$ implies that $\langle p_k, l\rangle \in W_i^{f_i}$. □

Alg. 3 Adaptive long-lived immediate snapshot: code for process p_i.

local variables:
  V', V, U, W : view
  $A_i[0,\ldots,N]$: array of views, initially $\perp$ // persistent
  f, *start-level* : integer

view im-upscan(*count* : integer)
1. $V' = \mathsf{upscan}(count)$ // increment your counter and get a view
2. for all $id_j \in V'$ do // update views for other processes
3.   if $V'(id_j) > A_i[id_j](id_j)$ then // p_j updated its counter after the previous scan by p_i
4. $A_i[id_j] = V'$ // update the view containing the last counter of p_j
5. $V = \min\{A_j[id_i] \mid id_j \in V' \text{ and } A_j[id_i](id_i) = count\}$ // minimal view stored for p_i, which contains p_i's new counter
6. $f = \sum V$ // calculate start floor
7. $view[f] = V$ // write your initial view
8. $U = \mathsf{collect}_f()$ // collect id's of the processes registered in the start floor
9. $start\text{-}level = |U \cup V| + 1$ // estimate the number of participants in lower floors
10. while (**true**) do // descend through the floors $f-1, f-2, \ldots$
11. $f = f - 1$
12. $\mathsf{store}_f(\langle id_i, count\rangle)$ // register in floor f
13. $flag[f][i] = (view[f] \neq \perp)$
14. $W = \mathsf{os\text{-}im\text{-}upscan}_f(count,\ start\text{-}level)$
15.   if ($count > view[f](id_i)$ and for some $\langle id_j, c_j\rangle \in W$, $flag[f][j] == \mathbf{true}$) then
16.   return(join(W, $view[f]$)) // maximal counters appearing in W or $view[f]$

view procedure join(V_1, V_2 : view)
1. $\text{return}(\{\langle id_j, c_j\rangle \mid id_j \in V_1 \cup V_2 \text{ and } c_j == \max\{V_1(id_j), V_2(id_j)\}\})$

If process p_i returns V_i from floor f in its lth im-upscan, then $\langle p_i, l\rangle \in W_i^f \preceq V_i$, since os-im-upscan returns a snapshot. This proves the self-inclusion property.

The proof of the immediacy property follows [17, Lemma 3.3.6].

Lemma 10 (Immediacy). *The returned views satisfy the immediacy property.*

Proof. Assume that V_j, a view returned by p_j from floor f_j, includes the l-th value written by p_i. Let V_i be the view returned by the l-th im-upscan operation of p_i from floor f_i. We show that $V_i \preceq V_j$.

Assume that p_i returns from a floor above f_j (that is, $f_i > f_j$). Then $\langle id_i, l\rangle \notin view[f_j]$ (by the condition in Line 15) and $\langle id_i, l\rangle \notin W_j^{f_j}$ (since $\langle id_i, l\rangle$ does not participate in $\mathsf{os\text{-}im\text{-}upscan}_{f_j}$). Therefore, $\langle id_i, l\rangle \notin V_j$, which is a contradiction.

If p_i returns from a floor below f_j (that is, $f_i < f_j$), then by Lemma 9, $V_i \preceq V_j$.

If p_i returns from floor f_j, then $\langle id_i, l\rangle \notin view[f_j]$, implying that $\langle id_i, l\rangle \in W_j^{f_j}$. By the immediacy property of os-im-upscan in floor f_j, p_i gets a view $W_i^{f_j} \preceq W_j^{f_j}$. Since p_i and p_j read the same (non-$\perp$) view from $view[f_j]$, $V_i \preceq V_j$. $\square$

The next lemma completes the complexity analysis by bounding the number of floors a process accesses; its proof is similar to [17, Lemma 3.3.3].

Lemma 11. *In* $\mathsf{im\text{-}upscan}_i^l$, *process* p_i *descends through at most* k *floors.*

Proof. Process p_i starts in floor $\sum V$, where V is the minimal atomic snapshot view containing p_i's new value, which is stored for p_i by other processes (Line 5 of im-upscan). Since k processes are active, at most $k-1$ views are unwritten between V and the next (smallest) written view with p_i's previous value. Thus, p_i accesses at most k floors. $\square$

Since os-im-upscan in each floor requires $O(k^2)$ steps (see below), we have the next theorem:

Theorem 3. *Algorithm 3 solves the immediate snapshot problem, with* $O(k^3)$ *step complexity.*

5.4 One-shot Immediate Snapshot Algorithm

The one-shot immediate snapshot algorithm presented in this section follows Borowsky and Gafni [19]. In Algorithm 4, a process descends through *levels*, checking the levels of other processes, until the number of processes in the levels below is larger than the level. In our algorithm, processes may start at different levels; however, as proved above, every process starts os-im-upscan on floor f at a level larger than k_f, the number of the processes accessing floor f.

The set of processes *descending* to level ℓ, by performing $\mathsf{store}(\langle *, *, \ell\rangle)$ after $\mathsf{store}(\langle *, *, \ell+1\rangle)$, is denoted D_ℓ. At most ℓ processes descend to level ℓ [17, Lemma 3.1.1].

Lemma 12. $|D_\ell| \leq \ell$, *for every level* ℓ, $1 \leq \ell \leq n$.

Proof. Assume, by way of contradiction, that $\ell + 1$ (or more) processes descend to level ℓ. Let p_j be the process in D_ℓ whose $\mathsf{store}(\langle *, \ell+1\rangle)$ is the latest to com-

Alg. 4 One-shot immediate snapshot (based on [19]): code for process p_i.

procedure os-im-upscan(*count*, *start-level* : integer)
    returns a view
$level = start\text{-}level$
store($\langle id_i, count, level\rangle$) // the start level of p_i
while (**true**) do
    $level = level - 1$
    store($\langle id_i, count, level\rangle$) // p_i descends one level
    V = collect()
        // returns a set of $\langle id, counter, level\rangle$ triples
    $W = \{\langle id_j, count_j, level_j\rangle \in V \mid level_j \leq level\}$
        // processes on smaller or equal levels
    if ($|W| \geq level$) then return(W)

plete. Since processes' levels do not increase, p_j's following collect returns at least $\ell + 1$ processes in levels $1, \ldots, \ell + 1$, and p_j does not descend to level ℓ. □

If p_i starts at a level larger than k, then it descends to level $\ell \leq k$ after descending to levels $k, \ldots, \ell + 1$; thus, if $p_i \in D_\ell$, then $p_i \in D_k, \ldots, D_{\ell+1}$, implying the next lemma:

Lemma 13. *If all process start above level k, then $D_1 \subseteq D_2 \subseteq \ldots \subseteq D_k$.*

Let S_i be the set of processes in the view p_i returns from some level ℓ; S_i contains only processes descending to level ℓ or below. By Lemma 13, $S_i \subseteq D_\ell$ and by Lemma 12, $|D_\ell| \leq \ell$. By the algorithm, $\ell \leq |S_i|$, which implies the next lemma:

Lemma 14. *If all processes start above level k, then $S_i = D_\ell$.*

If process p_i returns from level l_i, $1 \leq l_i \leq k$, then $p_i \in D_{l_i}$ which is equal to S_i (by Lemma 14); thus, the returned views satisfy the self-inclusion property.

If another process p_j returns from level l_j, then Lemmas 13 and 14 imply that either $S_i \subseteq S_j$ (if $l_i \leq l_j$) or $S_j \subseteq S_i$ (if $l_j \leq l_i$); thus, the returned views are comparable.

If $p_i \in S_j$, then $p_i \in D_{l_j}$, by Lemma 14. That is, p_i descends to level l_j and hence, $l_i \leq l_j$. By Lemmas 13 and 14, $S_i = D_{l_i} \subseteq D_{l_j} = S_j$, implying the immediacy property.

When called from Algorithm 3, process p_i descends through at most $start\text{-}level_i \leq k+1$ levels. In each level, it performs $O(k)$ operations (using our store and collect procedures), implying the next theorem:

Theorem 4. *If all processes start above level k, Algorithm 4 solves the one-shot immediate snapshot problem with $O(k^2)$ step complexity.*

6 Adaptive $(2k-1)$-Renaming

The (one-shot) $(2k-1)$-*renaming* problem [10] requires processes to acquire distinct names in the range $\{0, \ldots, 2k-2\}$. The algorithm of Borowsky and Gafni [19], can be made adaptive by using our immediate snapshot algorithm. The BG renaming algorithm proceeds in rounds; a process takes an immediate snapshot in each round, and processes are partitioned into groups according to the size of the returned views. The views also partition the name space into disjoint intervals; processes in each group continue the algorithm in the associated interval. The process with the maximal id in the group gets a name in the interval; other processes proceed to the next round. The code appears in Algorithm 5; a process starts the algorithm by calling rename(0,**true**).

For simplicity of presentation, $2n-1$ distinct immediate snapshot objects are associated with slots $0, 1, \ldots 2n-2$. In the first round, starting from slot 0, adaptive immediate snapshot Algorithm 3 is used, since the number of participating processes is not known. In later rounds, starting from slots $1, \ldots, 2n-2$, the size of the group is bounded by the size of the view obtained in the first round, therefore it suffices to use non-adaptive Algorithm 4 with appropriate parameter *start-level*.

As proved in [19], at least one process halts in each round; therefore, the number of rounds is at most k. In the first round, Algorithm 3 requires $O(k^3)$ steps, while in each of the later rounds, Algorithm 4 requires $O(k^2)$ steps. This implies the next theorem:

Theorem 5. *Algorithm 5 solves the one-shot $(2k-1)$-renaming problem, with $O(k^3)$ step complexity.*

7 Discussion

This paper presents an adaptive collect algorithm; the algorithm is simple and its step complexity is linear in the number of active processes. Many algorithms can be made adaptive by substituting our collect algorithm. In particular, we show how to obtain adaptive algorithms for atomic snapshots, with $O(k \log k)$ step complexity, for immediate snapshots, with $O(k^3)$ step complexity, and for $(2k-1)$-renaming problem, with $O(k^3)$ step complexity.

An adaptive *long-lived* $(2k-1)$-renaming algorithm can easily be derived from the ℓ-assignment algorithm of Burns and Peterson [20], using our collect algorithm. However, the step complexity of the resulting algorithm is at least exponential in k, since the step complexity of Burns and Peterson's algorithm is at least exponential in n [23]. A polynomial long-lived $(2k-1)$-renaming algorithm, which adapts to the current contention, appears in [12].

Acknowledgments: We thank Yehuda Afek, Yossi Levanoni, Dan Touitou and Gideon Stupp for helpful discussions.

References

1. Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, Sept. 1993.

Alg. 5 Adaptive $(2k-1)$-renaming (based on [19]): code for process p_i.

```
shared objects:
    Slots[0] : an adaptive immediate snapshot object              // Algorithm 3, used here as one-shot
    Slots[1 ... 2n - 2] : array of immediate snapshot objects    // Algorithm 4
local variables:
    S : view, initially ⊥
    firstSlot : integer, initially 0                              // first slot of the current range
    start-level : integer, initially 0
    direction : Boolean, initially true

rename(firstSlot, direction)
    if ( firstSlot == 0 ) then                                   // only the first round starts from slot 0
        S = Slots[0].im-upscan(1)                                // Algorithm 3
        start-level = |S|                                        // estimate the number of processes
    else S = Slots[firstSlot].os-im-upscan(1, start-level)       // Algorithm 4
    firstSlot = NewInterval(firstSlot, direction, |S|)
    if ( id_i is the maximal id in S ) then return(firstSlot)    // allocate self
    else rename(firstSlot, ¬direction)                           // start again in a new range, opposite direction

NewInterval(slot, direction, snapSize)       // allocate an interval of name space for the processes in the group
    if ( direction ) then return(slot + (2snapSize - 1))         // allocate going up
    else return(slot - (2snapSize - 1))                          // allocate going down
```


2. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proc. 18th ACM Symp. Principles of Dist. Comp.*, pages 91–103, 1999.
3. Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proc. 27th ACM Symp. Theory of Comp.*, pages 538–547, 1995.
4. Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proc. 18th ACM Symp. Principles of Dist. Comp.*, pages 105–112, 1999.
5. Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive collect with applications. In *Proc. 40th IEEE Symp. Foundations of Comp. Sci.*, pages 262–272, 1999.
6. Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot. In *Proc. 19th ACM Symp. Principles of Dist. Comp.*, pages 71–80, 2000.
7. Y. Afek and E. Weisberger. The instancy of snapshots and commuting objects. *J. Alg.*, 30(1):68–105, Jan. 1999.
8. R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the Real-Time Systems Symposium*, pages 12–22, Dec. 1992.
9. J. H. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementation. *Dist. Comp.*, 11(1):1–20, 1997.
10. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, July 1990.
11. H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proc. 17th ACM Symp. Principles of Dist. Comp.*, pages 277–286, 1998.
12. H. Attiya and A. Fouren. Polynomial and adaptive long-lived (2k-1)-renaming. In *Proc. 14th Int. Symp. on Dist. Computing*, pages 149–163, 2000.
13. H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, to appear. An extended abstract appeared in [11].
14. H. Attiya, M. Herlihy, and O. Rachman. Atomic snapshots using lattice agreement. *Dist. Comp.*, 8(3):121–132, 1995.
15. H. Attiya and O. Rachman. Atomic snapshots in $O(n \log n)$ operations. *SIAM J. Comput.*, 27(2):319–340, Mar. 1998.
16. H. Attiya and S. Rajsbaum. The combinatorial structure of wait-free solvable tasks. In *Proc. 10th Int. Workshop on Dist. Algorithms*, number 1151 in Lecture Notes in Computer Science, pages 321–343. Springer-Verlag, 1996. Also Technical Report #CS0924, Department of Computer Science, Technion, December 1997.
17. E. Borowsky. *Capturing the Power of Resiliency and Set-Consensus in Distributed Systems.* PhD thesis, Department of Computer Science, UCLA, 1995.
18. E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proc. 25th ACM Symp. Theory of Comp.*, pages 91–100, 1993.
19. E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proc. 12th ACM Symp. Principles of Dist. Comp.*, pages 41–52, 1993.
20. J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *Proc. 8th ACM Symp. Principles of Dist. Comp.*, pages 145–158, 1989.
21. M. Choy and A. K. Singh. Adaptive solutions to the mutual exclusion problem. *Dist. Comp.*, 8(1):1–17, 1994.
22. C. Dwork, M. Herlihy, and O. Waarts. Bounded round numbers. In *Proc. 12th ACM Symp. Principles of Dist. Comp.*, pages 53–64, 1993.
23. A. Fouren. Exponential examples for two renaming algorithms. www.cs.technion.ac.il/~hagit/pubs/expo.ps.gz, Aug. 1999.
24. A. Fouren. *Adaptive Wait-Free Algorithms for Asynchronous Shared-Memory Systems.* PhD thesis, Department of Computer Science, The Technion, June 2001.
25. M. Herlihy. Wait-free synchronization. *ACM Trans. Prog. Lang. Syst.*, 13(1):124–149, Jan. 1991.
26. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, Feb. 1987.
27. M. Moir. Fast, long-lived renaming improved and simplified. *Sci. Comput. Programming*, 30(3):287–308, May 1998.

28. M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Programming*, 25(1):1–39, Oct. 1995.
29. M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, 2000.

Hagit Attiya received the B.Sc. degree in Mathematics and Computer Science from the Hebrew University of Jerusalem, in 1981, the M.Sc. and Ph.D. degrees in Computer Science from the Hebrew University of Jerusalem, in 1983 and 1987, respectively. She is presently an associate professor at the department of Computer Science at the Technion, Israel Institute of Technology. Before joining the Technion, she has been a post-doctoral research associate at the Laboratory for Computer Science at M.I.T.

Arie Fouren recieved his M.A. in Computer Science from the Technion, Haifa, on 1999. He has just completed a Ph.D. in Computer Science, at the Technion.

Eli Gafni is currently an Associate Professor at UCLA computer Science Department. He received is B.A. from the Technion in 1972 and M.Sc. and Ph.D. from University of Illinois at Urbana-Champaign, and M.I.T., in 1979, and 1982, respectively.