An Adaptive Collect Algorithm with Applications

Hagit Attiya\(^1\), Arie Founen\(^1\), 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
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 $O(k^3)$ step complexity.

Several papers [9, 27, 28] study algorithms whose step complexity depends only on $n$, and not on the range of processes' 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) testset 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 $O(k^3)$ 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 $\alpha$ 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 $\phi_r$, 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 $\alpha$ if it takes a step in $\alpha$. Let $k(\alpha)$ be the number of active processes in $\alpha$.

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

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

Algorithm A is adaptive (to total contention) if there is a function $f: \mathbb{N} \rightarrow \mathbb{N}$ such that the following holds for every execution $\alpha$ of A: if process $p_i$ has an operation interval $\beta$ in $\alpha$, 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 $\alpha$.

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, $\{ (p_i, v_i), \ldots \}$, without repetitions of processes. $V(id_j)$ refers to $v_j$, if $(p_j, v_j) \in V$, and to 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 $V(p_j) = -$, then no store operation of $p_j$ precedes $cop$, if $V(p_j) = v \neq -$ 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 $(p_i, v_i) \in V_1$, we have $(p_i, v_i) \in V_2$, and $v_i$ is written in a store operation of $p_i$ that follows or is equal to a store operation of $p_i$ that writes $v_i'$. 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.
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 \( \text{stop} \), \( \text{left} \), or \( \text{right} \). It is guaranteed that if a single process enters the splitter, then it obtains \( \text{stop} \), 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 \( \text{store} \), a process acquires a vertex \( v \); from this point on, the process stores its up-to-date values in \( v.\text{val} \).

A process acquires a vertex in the tree using procedure \( \text{register} \); in \( \text{register} \), 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 \( \text{left} \), it moves to the left child; if it receives \( \text{right} \), it moves to the right child. A process marks each vertex it accesses by raising a flag associated with the vertex; a vertex is \emph{marked}, if its flag is raised. The process acquires a vertex \( v \) when it obtains \( \text{stop} \) at the splitter associated with \( v \); then it writes its id into \( v.\text{id} \). (See Figure 2.)

To perform a \( \text{collect} \), 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, \( X = - \) and \( Y = \text{false} \). A process executing the splitter first writes its id into \( X \) and then reads \( Y \). If \( Y = \text{true} \), then the process returns \( \text{right} \). Otherwise, the process sets \( Y = \text{true} \) and checks \( X \). If \( X \) still contains its id, then the process returns \( \text{stop} \); if \( X \) does not contain its id, then the process returns \( \text{left} \). The following lemma, from [28], states the main properties of the splitter.

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

The code for \text{collect} and \text{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:
- \( \text{mark} \): Indicates whether some process accessed \( v \); initially \text{false}.
- \( \text{id} \): Holds the identifier of the process that stops in \( v \); initially \(-\).
- \( \text{value} \): Holds an updated value of the process that stops in \( v \); initially \(-\).
- \( X \): Holds a process’ identifier, for the splitter associated with \( v \); initially \(-\).
- \( Y \): Holds a Boolean value, for the splitter associated with \( v \); initially \text{false}.
- \( \text{left-child} \): Pointer to the left child of \( v \).
- \( \text{right-child} \): Pointer to the right child of \( v \).

To prove the correctness and complexity of the algorithm, fix an execution \( o \) of the algorithm and let \( k \) be the number of processes that call \text{store} at least once in \( o \).

**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 \( u \) 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 \( \text{left} \) 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 \( \text{right} \) at \( u \) and access \( v \). \( \square \)

By Lemma 2 and the algorithm, when a process performs \( \text{register} \), 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
Lemma 4. All vertices on the path from the root to a marked vertex \(v\) are marked.

Assume that \(cop\) is a \textit{collect} of process \(p_i\). If some process \(p_j\) completes its first \textit{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 \textit{store} of \(p_j\) that follows \(cop\). Moreover, since values are updated with a single operation, a later \textit{collect} returns a more up-to-date view. This implies that Algorithm 1 solves the collect problem.

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

The step complexity of \textit{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 \textit{collect} operation, and let \(\alpha\) be the shortest finite execution prefix that contains its execution interval. Let \(S = v_0, v_1, \ldots, v_t\) be the DFS traversal of the marked tree after \(\alpha\), appearing in an in-order; i.e., for every marked vertex \(v\), the vertices of the left subtree 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 \textit{grey} if there is a process that accesses the left child of \(v\) in \(\alpha\) (that is, writes \textit{true} in the variable \textit{mark} of \(v\text{-left-child}\)) and there is a process that accesses the right child of \(v\) in \(\alpha\) (that is, writes \textit{true} in the variable \textit{mark} of \(v\text{-right-child}\)). A marked vertex that is not grey is \textit{black}. For any black vertex \(v\), (a) there is a process that sets \(v\text{-mark}\) to \textit{true}, and (b) one of the children of \(v\) (e.g., \(v\text{-right-child}\)) is not accessed by any process in \(\alpha\). By Lemma 1(2), not all the processes accessing \(v\) return \textit{left}. Therefore, at least one process that accesses \(v\) either returns \textit{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, \(v_{i+1}\) 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 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 \( a \); 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 \( \text{collect} \) 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 \( t \)th upscan operation of \( p_j \) includes the \( t \)th value written by \( p_j \).

**Comparability:** If \( V_1 \) and \( V_2 \) are the views returned by two upscan operations, then either \( V_1 \subseteq V_2 \) or \( V_2 \subseteq V_1 \).

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 non-adaptive algorithm, processes descend down a binary tree of depth \( \Omega(\log n) \) (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, ... 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 \( \text{Label}(v) \) 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. \( \left| \bigcup\{O_j \mid p_j \text{ returns left}\} \right| \leq M \), and
2. if \( p_i \) returns right, then \( \bigcup\{O_j \mid p_j \text{ returns left}\} \) contains \( O_i \).

The adaptive algorithm uses an unbalanced binary tree, constructed from \( \log n \) complete binary trees of exponentially growing sizes (1, 2, 2\(^2\), ... leaves), connected by a single path (Figure 4). As in the non-adaptive algorithm, leaves are labeled 1, 2, ... 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.
The algorithm implies that \( V_j \) is contained in \( \cup \{ O_t \mid \rho_t \text{ returns left} \} \) and \( O_t \) 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 \( M = \text{Label}(v) \). Let \( u \) be the last vertex (on the path from the root to \( v \)) in which \( op_i \) and \( op_j \) go right; \( r \) is the left-most leaf in the right subtree of \( u \) and by construction, \( \text{Label}(u) = M - 1 \).

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; \( r \) is the right-most leaf in the left subtree of \( w \) and hence, \( \text{Label}(w) = \text{Label}(w) = M \). As mentioned above, the algorithm implies that \( V_i \) and \( V_j \) are contained in \( \cup \{ O_t \mid \rho_t \text{ returns left and } O_t \text{ in } v \} \).

By Lemma 7(1), this union contains at most \( \text{Label}(w) \) 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 \text{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 \( \text{im-upscan} \) 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 \( \text{im-upscan} \) operation, \( V_i \), includes the value written in the \( i \)th \( \text{im-upscan} \) of \( p_j \) that returns the view \( V_j \), then \( V_j \leq V_i \).

#### 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 \( \text{im-upscan} \) 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 \textit{sum} of counters in a view \( V \), denoted \( \Sigma V \), is the total number of updates preceding \( V \); clearly, for views satisfying the comparability property \( \Sigma V_1 \leq \Sigma V_2 \) if and only if \( V_1 \leq V_2 \).

The overall structure of our algorithm follows the non-adaptive immediate snapshot algorithm [7, 17]. Process \( p_k \) first finds an atomic snapshot view \( V \) containing its new value. \( V \) is written in a floor whose number \( s_1 \) is equal to \( \Sigma V \); clearly, all views written in the same floor are equal. (The number of floors is infinite, since \( \Sigma V \) is unbounded for a long-lived algorithm.) Then, \( p_k \) participates in a distinct copy of one-shot immediate snapshot in each floor below \( s_1 \), with its new value as input, until it sees its previous value in the view written in one of these floors. When this happens, \( p_k \) 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 \( p_k \) accesses, it takes as \( V \) the \textit{smallest} atomic snapshot view containing its new value among the snapshots obtained by other processes. That is, \( s_k \) is the smallest floor where a view containing \( p_k \)'s new value is written. This is used below (Lemma 11) to show that \( p_k \) accesses at most \( k \) floors. To allow \( p_k \) to find the smallest view containing its new value, each process \( p_j \) maintains an array which holds, for every process \( p_k \), the first view \( p_j \) observes with the most recent value of \( p_k \); \( p_j \) updates this view whenever it sees a new value for \( p_k \). To find \( V \) and calculate its start floor, \( p_k \) reads the appropriate entries of the active processes' arrays and picks the minimal view containing its last value.

Since the view written in floor \( s_1 \) contains \( p_k \)'s new value, processes returning from floors \( \geq s_1 \) see this (or a later) value of \( p_k \). Since \( p_k \) returns from some floor \( \text{floor} < s_1 \) containing its previous value, processes returning from floors \( < \text{floor} \) see previous values of \( p_k \). Since \( p_k \) performs the one-shot immediate snapshot algorithm with its last value as input in each floor between \( s_1 \) and \( \text{floor} \), the views returned from these floors include this value. The one-shot immediate snapshot in floor \( \text{floor} \) guarantees that views returned from floor \( \text{floor} \) 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-
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 \( \text{os-im-upscan}_f \), is associated with every floor \( f \), as well as the following data structures:

1. \( \text{view}[f] \), a view; initially contains the empty view, \(-\).
2. \( \text{flag}[f][1 \ldots N] \) 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; \( A_i[id_i] \) holds the first view containing the last value of \( p_i \), 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 im-upscan operation. If it did, then \( p_i \) writes \( V' \) (containing the new counter of \( p_j \)) into \( A_i[id_i] \). 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 \( A_i[id_i] \) 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 below its start floor. In each floor \( f \), if \( p_i \) reads a non-\(-\) view from \( \text{view}[f] \) then \( p_i \) sets \( \text{flag}[f][i] \) to true. Then, \( p_i \) obtains a view \( W_i^f \) from \( \text{os-im-upscan}_f \). If \( \text{view}[f] \) 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 \( \text{view}[f] \); 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 \( \text{store} \) and \( \text{collect} \) (Algorithm 1), denoted \( \text{store}_f \) and \( \text{collect}_f \), is associated with each floor \( f \). A process registers before accessing floor \( f \), using \( \text{store}_f \). Process \( p_i \) collects a set \( U_i \) of processes registered in its start floor, \( s_i \), using \( \text{collect}_{s_i} \). \( |V_i \cup U_i| + 1 \) is the start level parameter of \( p_i \) for \( \text{os-im-upscan} \) 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 \( \text{im-upscan} \) by processes \( p_i \) do not call the same copy of \( \text{os-im-upscan} \). If an operation \( op_1 \) of \( p_i \) starts in floor \( f \), then \( op_1 \) calls \( \text{os-im-upscan} \) 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 \( \text{os-im-upscan} \) 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-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 \( \text{store}_{s_i} \) before \( p_i \) starts \( \text{collect}_{s_i} \), then \( p_j \in U_i \) and the lemma follows.

Otherwise, \( p_j \) reads \( V_i \neq - \) from \( \text{view}[s_i] \) since it reads after completing \( \text{store}_{s_i}(id_j) \), and \( p_j \) writes \( V_i \) into \( \text{view}[s_i] \) before starting \( \text{collect}_{s_i} \). 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 \( p_j \) returns \( V_i \) and \( p_j \) returns \( V_j \) from the same floor \( f \), then they read the same value from \( \text{view}[f] \). \( W_i^f \) and \( W_j^f \) are views returned by \( \text{os-im-upscan}_f \) 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, \( \text{view}[f_i] \preceq \text{view}[f_j] \). We show that \( W_i^{f_j}(p_k) \leq V_i(p_k) \), for any process \( p_k \).

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

If \( s_k \geq f_j \), 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-\(-\) value from \( \text{view}[f_i] \) that includes \( (p_k,l) \), then \( (p_k,l) \in \text{view}[f_i] \Rightarrow \text{view}[f_i] \preceq V_i \), since \( p_k \) reads the same non-\(-\) value from \( \text{view}[f_i] \).

Otherwise, \( p_k \) reads \( \text{false} \) from \( \text{flag}[f_i][l] \), for every process \( p_k \in W_i^{f_j} \). Clearly, \( p_k \) reads \( \text{true} \) from \( \text{flag}[f_i][y] \), for some process \( p_y \in W_i^{f_j} \). However, \( p_y \) writes \( \text{true} \) to \( \text{flag}[f_i][y] \) before calling \( \text{os-im-upscan} \), and \( p_k \) must read \( \text{true} \) from \( \text{flag}[f_i][y] \) if \( p_y \in W_i^{f_j} \). Thus, \( p_k \notin W_i^{f_j} \), and by the comparability property, \( W_i^{f_j} \preceq W_i^{f_j} \). The self-inclusion property of \( \text{os-im-upscan} \) implies that \( (p_k,l) \in W_i^{f_j} \).
Alg. 3 Adaptive long-lived immediate snapshot: code for process $p_i$.

local variables:

$V', V, U, W : \text{view}$
$A_i[0, \ldots, N] : \text{array of views, initially} -$
$f, \text{start-level} : \text{integer}$ // persistent

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

If process $p_i$ returns $V_i$ from floor $f$ in its $l$th im-upscan, then $(p_i, l) \in W_f^l \subseteq V_i$, 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 $V_i \sqsubseteq V_j$.

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

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

If $p_i$ returns from floor $f_j$, then $\langle id, l \rangle \not\in \text{view}[f_j]$, implying that $\langle id, l \rangle \not\in W_{f_j}^l$. By the immediacy property of os-im-upscan in floor $f_j$, $p_i$ gets a view $W_{f_j}^l \supset W_{f_j}^j$. Since $p_i$ and $p_j$ read the same (non-) view from $\text{view}[f_j]$, $V_i \sqsubseteq V_j$. □

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 im-upscan$_l$, 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. □

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 [9]. 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 $\ell$, by performing store$(s, s, \ell)$ after store$(s, s, \ell + 1)$, is denoted $D_\ell$. At most $\ell$ processes descend to level $\ell$ [17, Lemma 3.1.1].

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

Proof. Assume, by way of contradiction, that $\ell + 1$ (or more) processes descend to level $\ell$. Let $p_j$ be the process in $D_\ell$ whose store$(s, s, \ell + 1)$ is the latest to com-
Algorithm 4: One-shot immediate snapshot (based on [19]):

procedure os-im-upscan(count, start-level: integer)
returns a view
1. level = start-level
2. store((id, count, level)) // the start level of pi
3. while (true) do
4. level = level + 1
5. store((id, count, level)) // pi descends one level
6. V = collect() // returns set of (id, counter, level) triples
7. W = {(id, counter, level) ∈ V | level_j ≤ level}
8. if (|W| ≥ 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 \( \ell_j \) does not descend to level \( \ell \). □

If \( p_k \) starts at a level larger than \( k \), then it descends to level \( \ell \) after descending to levels \( k, \ldots , \ell + 1 \); thus, if \( p_k \in D_\ell \), then \( D_k \subseteq D_\ell \subseteq \cdots \subseteq D_k \).

Lemma 13. If all process start above level \( k \), then \( D_1 \subseteq D_2 \subseteq \cdots \subseteq D_k \).

Let \( S_i \) be the set of processes in the view \( p_k \) returns from some level \( \ell \); \( S_i \) contains only processes descending to level \( \ell \) 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 \( p_k \) returns from level \( l_i \), \( 1 \leq l_i \leq k \), then \( p_k \in D_\ell \), 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 \( p_k \in S_j \), then \( p_k \in D_{l_j} \), by Lemma 14. That is, \( p_k \) 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 \( p_k \) descends through at most \( \text{start-level} \leq k + 1 \) 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 \( \ell \)-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

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

shared objects:
- \(\text{Slots}[0]\) : an adaptive immediate snapshot object
- \(\text{Slots}[1 \ldots 2n - 2]\) : array of immediate snapshot objects

local variables:
- \(S\) : view, initially
- \(\text{firstSlot}\) : integer, initially 0
- \(\text{start-level}\) : integer, initially 0
- \(\text{direction}\) : Boolean, initially true

\(\text{rename}(\text{firstSlot}, \text{direction})\)

1. if \((\text{firstSlot} = 0)\) then
   \[S = \text{Slots}[0].\text{im-upscan}(1)\]
   \(\text{start-level} = |S|\)
2. else \(S = \text{Slots}[\text{firstSlot}].\text{os-im-upscan}(1, \text{start-level})\)
3. \(\text{firstSlot} = \text{NewInterval}(\text{firstSlot}, \text{direction}, |S|)\)
4. if \((i.d. \text{is the maximal } i.d. \text{in } S)\) then return \(\text{firstSlot}\)
5. else \(\text{rename}(\text{firstSlot}, \neg \text{direction})\)
6. \(\text{if } \text{direction} \text{ then return } \text{slot} + (2 \times \text{snapSize} - 1)\)
7. else return \(\text{slot} - (2 \times \text{snapSize} - 1)\)

\(\text{NewInterval}(\text{slot, direction, snapSize})\)

1. if \((\text{direction})\) then return \(\text{slot} + (2 \times \text{snapSize} - 1)\)
2. else return \(\text{slot} - (2 \times \text{snapSize} - 1)\)

\[\text{Algorithm } 3\text{, used here as one-shot}\]
\[\text{Algorithm } 4\]


**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** received 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 a 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.