Efficient Adaptive Collect using Randomization*

Hagit Attiya[1] Fabian Kuhn[2] C. Greg Plaxton[3] Mirjam Wattenhofer[4]
Roger Wattenhofer[2]

Abstract

An *adaptive* algorithm, whose step complexity adjusts to the number of active processes, is attractive for distributed systems with a highly-variable number of processes. The cornerstone of many adaptive algorithms is an adaptive mechanism to collect up-to-date information from all participating processes. To date, all known collect algorithms either have non-linear step complexity or they are impractical because of unrealistic memory overhead.

This paper presents new randomized collect algorithms with asymptotically optimal $\mathrm{O}(k)$ step complexity and linear memory overhead only. In addition we present a new deterministic collect algorithm that beats the best step complexity for previous polynomial-memory algorithms.

1 Introduction and Related Work

To solve certain problems, processes need to collect up-to-date information about the other participating processes. For example, in a typical *indulgent* consensus algorithm [11, 12], a process needs to announce its preferred decision value and obtain the preferences of all other processes. Other problems where processes need to collect values are in the area of atomic snapshots [1, 3, 9], mutual exclusion [2, 4, 6, 7], and renaming [2]. A simple way that information about other processes can be communicated is to use an array of registers indexed by process identifiers. An active process can update information about itself by writing into its register. A process can collect the information it wants about other participating processes by reading the entire array of registers. This takes $\mathrm{O}(n)$ steps, where n is the total number of processes.

When there are only a few participating processes, it is preferable to be able to collect the required information more quickly. An *adaptive* algorithm is one whose step complexity is a function of the number of participating processes. Specifically, if it performs at most $h(k)$ steps when there are k participating processes, we say that it is h-adaptive. An algorithm is *wait-free* if all processes can complete their operations in a finite number of steps, regardless of the behavior of the other processes [13].

Several adaptive, wait-free collect algorithms are known [2, 8, 9]. In particular, there is an algorithm that features an asymptotically optimal $\mathrm{O}(k)$-adaptive collect, but its memory consumption is exponential

* A preliminary version of this paper appeared in the Proceedings of the 18th Annual Conference on Distributed Computing (DISC) 2004 [10].

[1] Department of Computer Science, The Technion, Haifa 32000, Israel. Email: hagit@cs.technion.ac.il.

[2] Department of Information Technology and Electrical Engineering, ETH Zurich, 8092 Zurich, Switzerland. Email: {kuhn,wattenhofer}@tik.ee.ethz.ch.

[3] Department of Computer Science, University of Texas at Austin, 1 University Station C0500, Austin, Texas 78712–0233. Email: plaxton@cs.utexas.edu. Partially supported by NSF Grants CCR–0310970 and ANI–0326001. Also affiliated with Akamai Technologies, Inc., Cambridge, MA 02142.

[4] Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland. Email: mirjam.wattenhofer@inf.ethz.ch.

in the number of potential processes [9], which renders the algorithm impractical. Other algorithms have polynomial (in the number of potential processes) memory complexity, but the collect costs $\Theta(k^2)$ steps [9, 16]. (Moir and Anderson [16] employ a matrix structure to solve the renaming problem. The same structure can be used to solve the collect problem, following ideas of [9].) The lower bound of Jayanti, Tan and Toueg [14] implies that the step complexity of a collect algorithm is $\Omega(k)$. This raises the question of the existence of a collect algorithm that features an asymptotically optimal $\mathrm{O}(k)$ step complexity and needs polynomial memory size only.

This paper suggests that randomization can be used to make adaptive collect algorithms more efficient, in contrast to known deterministic algorithms with either super-linear step complexity or unrealistic memory overhead. We present a wait-free randomized algorithm with memory complexity that is linear in n, step complexity that is linear in k for the collect operation, and step complexity that is nearly logarithmic in k for the first invocation of a store operation. The algorithm is randomized, and the step complexity bounds hold "with high probability" as well as "in expectation." We believe that randomization may bring a fresh approach to the design of adaptive shared-memory algorithms.

Analogously to previous approaches, our randomized algorithm (Section 4) uses *splitters* as introduced by Moir and Anderson to govern the algorithmic decisions of processes [16]. A splitter has an associated register. Various processes may visit the splitter to try to acquire this register. The splitter ensures that at most one process succeeds in acquiring the register. In addition, the splitter partitions the unsuccessful processes into two sets. Ideally, these two sets are equal, or approximately equal, in size. Using a deterministic splitter, it is difficult to partition the unsuccessful processes into two approximately equal-sized sets. That being the case, it is natural to consider a *randomized* splitter that flips a fair coin to assign each unsuccessful process to one of the two output partitions. As in the deterministic linear collect algorithm of [9], where deterministic splitters are organized in a complete binary tree, we find it useful to study the behavior of a complete binary tree of randomized splitters, which we refer to as a randomized splitter tree. The randomized splitter tree is the basic building block of our randomized adaptive collect algorithm. The algorithm itself corresponds to a cascaded sequence of randomized splitter trees of geometrically decreasing size, followed by a deterministic backup structure. We prove that with high probability the backup structure is unused.

A binary tree of randomized splitters was previously used by Kim and Anderson [15] for adaptive mutual exclusion.

In addition, Section 3 introduces a new wait-free, deterministic algorithm that improves the trade-off between collect time and memory complexity: Using polynomial memory only, we achieve $o(k^2)$ collect. For any integer $\gamma > 1$, the algorithm provides a STORE with $\mathrm{O}(k)$ step complexity, a COLLECT with $\mathrm{O}(k^2/((\gamma-1)\log n))$ step complexity and $\mathrm{O}(n^{\gamma+1}/((\gamma-1)\log n))$ memory complexity. Interestingly, by choosing γ accordingly, our deterministic algorithm achieves the bounds of both previously known algorithms [9, 16].

All new algorithms build on the basic collect algorithm on a binary tree [9]. To employ this algorithm in a more versatile manner than its original design, we rely on a new and simplified proof for the linear step complexity of COLLECT (Section 3.1).

2 Model

We assume a standard asynchronous shared-memory model of computation. A system consists of n *processes*, $p_1, \ldots, p_n$, communicating by reading from and writing to shared *registers*.

Processes are state machines, each with a (possibly infinite) set of local states, which includes a unique *initial state*. In each *step*, the process determines which operation to perform according to its local state, and

subsequently changes its local state according to the value returned by the operation.

A *register* provides two operations: *read*, returning the value of the register; and *write*, changing the register value to the value of its input. A *configuration* consists of the states of the processes and the values of the registers. In the *initial configuration*, every process is in the initial state and all registers are $\perp$. A *schedule* is a (possibly infinite) sequence $p_{i_1}, p_{i_2}, \ldots$ of process identifiers. An *execution* consists of the initial configuration and a schedule, representing the interleaving of steps by processes.

An *implementation* of an object of type X provides for every operation *OP* of X a set of n procedures $F_1, \ldots, F_n$, one for each process. (Typically, the procedures are the same for all processes.) To execute *OP* on X, process p_i calls procedure F_i. The worst-case number of steps performed by some process p_i executing procedure F_i is the *step complexity* of implementing *OP*.

An operation OP_i *precedes* operation OP_j (and OP_j *follows* operation OP_i) in an execution α, if the call to the procedure of OP_j appears in α after the return from the procedure of OP_i.

Let α be a finite execution. Process p_i is *active* at the end of α if α includes a call of a procedure F_i without a matching return.

The *total contention* during α is the number of all processes that are active at the end of some prefix of α. Let f be a non-decreasing function. An implementation is *f-adaptive* to total contention if the step complexity of each invocation of its procedures in α is bounded from above by $f(k)$, where k is the total contention during α.

For completeness, we also define the stronger notion of adaptivity to point contention, which is not addressed in this paper. The *point contention* during an interval in α is the maximum number of processes that were simultaneously active at some point in time during that interval. An implementation is *f-adaptive to point contention* if the step complexity of its procedures is bounded by $f(k)$, where k is the point contention during the interval of the procedure.

A *collect algorithm* provides two operations: A STORE(*val*) by process p_i sets *val* to be the latest value for p_i. A COLLECT operation returns a *view*, a partial function V from the set of processes to a set of values, where $V(p_i)$ is the latest value stored by p_i, for each process p_i. A COLLECT operation *cop* should not read from the future or miss a preceding STORE operation *sop*. Formally, the following validity properties hold for every process p_i:

- If $V(p_i) = \perp$, then no STORE operation by p_i precedes *cop*.
- If $V(p_i) = v \neq \perp$, then v is the value of a STORE operation *sop* of p_i that does not follow *cop*, and there is no STORE operation by p_i that follows *sop* and precedes *cop*.

3 Deterministic Adaptive Collect

3.1 The Basic Binary Tree Algorithm

Associated to each vertex in the complete binary tree of depth $n-1$ is a *splitter* [16]: 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. Thus the set of processes is "split" into smaller subsets, according to the values obtained.

To perform a STORE in the algorithm of [9], a process writes its value in its acquired vertex. In case it has no vertex acquired yet it starts at the root of the tree and moves down the data structure according to the values obtained in the splitters along the path: If it receives a **left**, it moves to the left child, if it receives a

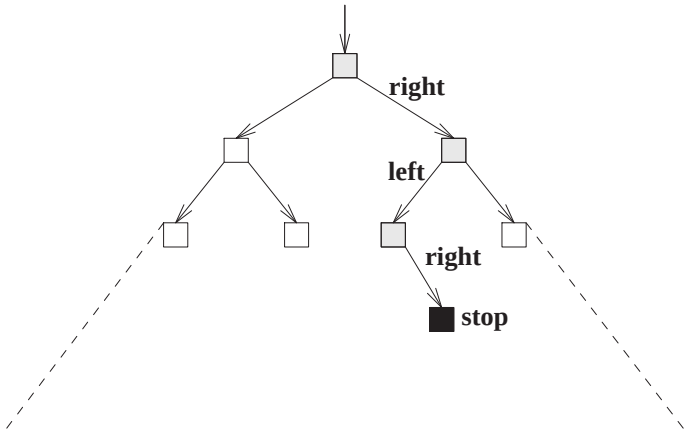

Figure 1: Traversing the basic binary tree.

right, it moves to the right child. A process marks each vertex it accesses by raising a flag associated with the vertex. Figure 1 illustrates how a process traverses the basic binary tree in a STORE operation.

We call a vertex *marked*, if its flag is raised. A process i acquires a vertex v, or stops in v, if it receives a **stop** at v's splitter. It then writes its id into $v.id$ and its value in $v.value$. In Figure 1, a vertex is black if it is acquired by some process, it is grey if it is marked, and white in all other cases. In later invocations of STORE, process i immediately writes its value in $v.value$, clearly leading to a constant step complexity. This leaves us to determine the step complexity of the first invocation of STORE.

In order 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 complete binary tree of depth $n-1$ has 2^n-1 vertices, implying the following lemma.

Lemma 3.1. *The memory complexity is* $\Theta(2^n)$.

Lemma 3.2 ([9]). *Each process writes its id in a vertex with depth at most* $k-1$ *and no other process writes its id in the same vertex.*

Lemma 3.3. *The step complexity of* COLLECT *at most* $2k-1$.

Proof. In order to perform a collect, a process traverses the marked part of the tree. Hence, the step complexity of a collect is equivalent to the number of marked (visited) vertices.

Let x_k be the number of marked vertices in a tree, where k processes access the root. The splitter properties imply the following recursive equations:

$$x_k = x_i + x_{k-i-1} + 1, \qquad (i \geq 0) \tag{1}$$

$$x_k = x_i + x_{k-i} + 1, \qquad (i > 0) \tag{2}$$

Equation (1) holds if a process stops in the splitter; otherwise, Equation (2) holds.

We prove the lemma by induction; note that the lemma trivially holds for $k = 1$. For the induction step, assume the lemma is true for $j < k$, that is, $x_j \leq 2j-1$. Then we can rewrite Equation (1):

$$x_k \leq (2i-1) + (2(k-i-1)-1) + 1 \leq 2k-1$$

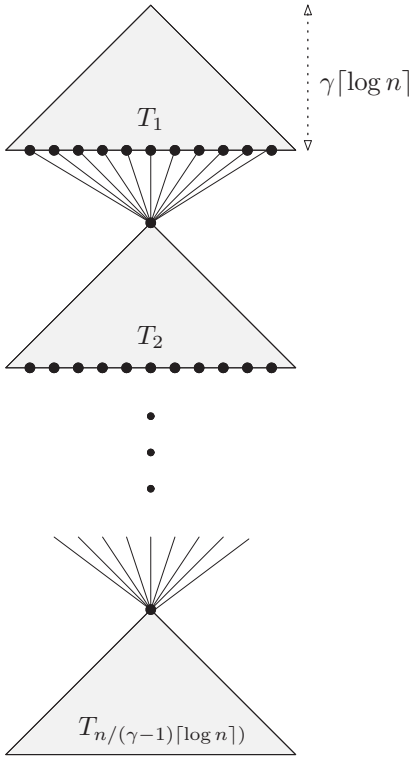

Figure 2: Organization of splitters in the cascaded trees algorithm.

and Equation (2) becomes:

$$x_k \leq (2i-1) + (2(k-i)-1) + 1 \leq 2k-1.$$

□

3.2 The Cascaded Trees Algorithm

We present a spectrum of algorithms, each providing a different trade-off between memory complexity and step complexity. For an arbitrary constant $\gamma > 1$, the *cascaded trees algorithm* provides a STORE with $\mathrm{O}(k)$ step complexity, a COLLECT with $\mathrm{O}(k^2/((\gamma-1)\log n))$ step complexity and $\mathrm{O}(n^{\gamma+1})$ memory complexity.

3.2.1 The Algorithm

The algorithm is performed on a sequence of $n/((\gamma-1)\lceil\log n\rceil)$ complete binary splitter trees of depth $\gamma\lceil\log n\rceil$, denoted $T_1, \ldots, T_{n/((\gamma-1)\lceil\log n\rceil)}$. (To keep the calculations simple, we assume that $\gamma\lceil\log n\rceil$ is an integer and that n is divisible by $(\gamma-1)\lceil\log n\rceil$.) Except for the last tree, each leaf of tree T_i has an edge to the root of tree T_{i+1} (Figure 2).

To perform a STORE, a process writes in its acquired vertex. If it has not acquired a vertex yet, it starts at the root of the first tree and moves down the data structure as in the binary tree STORE (described in the previous section). A process that does not stop at some vertex of tree T_i continues to the root of the next

Algorithm 1 Cascaded trees: Node acquisition

```
1: v = root of T_1
2: repeat
3:    v.mark = true
4:    move = splitter(v) {returns either stop, left, or right}
5:    if move == left then
6:       v = v.left-child
7:    else if move == right then
8:       v = v.right-child
9:    fi
10: until move == stop
11: v.id = id {write your identifier}
12: return(v)
```

tree. Note that both the right and the left child of a leaf in tree T_i, $1 \leq i \leq n/((\gamma-1)\lceil \log n \rceil) - 1$, are the root of the next tree. Algorithm 1 presents the code for acquiring a vertex in the cascaded trees; note that the code relies on the fact (proved below) that a process will obtain **stop** in one of the trees and does not include a condition to avoid "falling out" of the cascaded trees.

The splitter properties guarantee that no two processes stop at the same vertex.

To perform a COLLECT, a process traverses the part of tree T_i containing marked vertices in DFS order and collects the values written in the marked vertices. If any of the leaves of tree i are marked, the process also collects in tree T_{i+1}.

3.2.2 Analysis

We have $n/((\gamma-1)\lceil \log n \rceil)$ trees, each of depth $\gamma\lceil \log n \rceil$, implying the following lemma.

Lemma 3.4. *The memory complexity is*

$$\mathrm{O}\left(\frac{n^{\gamma+1}}{(\gamma-1)\log n}\right).$$

Let k be the number of processes that call STORE at least once and k_i be the number of processes that access the root of tree T_i.

Lemma 3.5. *At least* $\min\{k_i, (\gamma-1)\lceil \log n \rceil\}$ *processes do not exit from a leaf of tree* T_i *for every* i, $1 \leq i \leq n/(\gamma-1)\lceil \log n \rceil$.

Proof. Let m_i be the number of marked leaves in tree T_i. Consider the sub-tree T_i' that is induced by all the paths from the root to the marked leaves of T_i.

We first argue that a non-leaf vertex $v \in T_i'$ with one marked child in T_i' corresponds to at least one process that does not continue to T_{i+1}. If only one child value (**left** or **right**) is returned at v, then either some process obtained **stop** at v or some process did not return from the splitter associated with v. Otherwise, processes reaching v return both **left** and **right**. Since only one path leads to a leaf, say, the one through the left child, at least one process (that obtained **right** at v) does not access the right child of v and does not reach a leaf of T_i.

The number of vertices in T_i' with two children is exactly $m_i - 1$, since each node with two children adds one to the number of paths to the leaves in T_i'.

To count the number of vertices with one child, we estimate the total number of vertices in T_i' and then subtract $m_i - 1$.

Since T_i' is a subtree of a binary tree, the number of nodes at a level at most doubles the number of nodes in the preceding level. Conversely, the number of vertices on each preceding level is at least half the number at the current level. Starting above the leaves of T_i', whose number is m_i, we therefore get the following inequality for the number of non-leaf vertices n_i of tree T_i':

$$n_i \geq \underbrace{\frac{m_i}{2} + \frac{m_i}{4} + \cdots + \frac{m_i}{2^{\lceil \log m_i \rceil}}}_{m_i - 1} + \underbrace{1 + \cdots + 1}_{\gamma\lceil \log n \rceil - \lceil \log m_i \rceil},$$

where the number of ones in the equation follows from the fact that the tree T_i has depth $\gamma \log n$ and after $\lceil \log m_i \rceil$ levels the number of vertices on the preceding level is at least one. The claim follows since $m_i \leq n$. □

Lemma 3.6. *A process writes its id in a vertex in tree T_m at the latest, for the smallest m such that $k \leq m \cdot (\gamma - 1)\lceil \log n \rceil$.*

Proof. If $k \leq (\gamma - 1)\lceil \log n \rceil$, then a process stops in tree T_1, by Lemma 3.2, and the claim follows.

Assume $(m-1) \cdot (\gamma - 1)\lceil \log n \rceil < k \leq m \cdot (\gamma - 1)\lceil \log n \rceil$, for some integer $m > 1$. By Lemma 3.5 at least $(\gamma - 1)\lceil \log n \rceil$ processes do not exit from a leaf of tree T_i, for every i, $1 \leq i \leq m - 1$. Thus, at most $(\gamma - 1)\lceil \log n \rceil$ processes access tree T_m and by Lemma 3.2, a process stops in a vertex of tree T_m at the latest. □

Thus a process stops after accessing at most $\lceil k/((\gamma - 1)\lceil \log n \rceil) \rceil$ trees. Since the depth of each tree is $\gamma \lceil \log n \rceil$ and each splitter requires a constant number of operations, it follows that the step complexity of the first invocation of STORE is $\mathrm{O}(k/((\gamma - 1)\lceil \log n \rceil) \cdot \gamma \lceil \log n \rceil) = \mathrm{O}(\gamma/(\gamma - 1)k)$. All invocations thereafter require $\mathrm{O}(1)$ steps.

By Lemma 3.3, the time to collect in tree T_i is $2k_i - 1$. By Lemma 3.6, all processes stop after at most $k/((\gamma - 1)\log n)$ trees. This implies the next lemma:

Lemma 3.7. *The step complexity of a* COLLECT *is*

$$\mathrm{O}\left(\frac{k^2}{(\gamma - 1)\log n}\right).$$

Remark: The cascaded-trees algorithm provides a spectrum of trade-offs between memory complexity and step complexity. Choosing $\gamma = 1 + 1/\log n$ gives an algorithm with $O(k^2)$ step complexity for COLLECT and $O(n^2)$ memory complexity; this matches the complexities of the matrix algorithm [16]. Setting $\gamma = n/\log n + 1$ yields a single binary tree of height n; namely, an algorithm where the step complexity of COLLECT is linear in k but the memory requirements are exponential, as in the algorithm of [9].

4 Adaptive Collect with Randomized Splitters

The algorithm presented in this section uses another kind of splitter, described in Section 4.1, that makes a random choice in order to direct processes left and right. In Section 4.2 we analyze the behavior of a

Algorithm 2 Randomized Splitter

```
1: X = id_i
2: if Y then return randomly right or left
3: Y = true
4: if (X == id_i) then
5:     return stop
6: else
7:     return randomly right or left
8: fi
```

complete binary tree of such randomized splitters. In Section 4.3 we present our adaptive collect algorithm, which utilizes a cascaded sequence of randomized splitter trees. In Section 4.4 we analyze this algorithm. Our three main results are Theorem 4.10, which bounds the memory complexity of the algorithm, Theorem 4.18, which bounds the step complexity of the first invocation of STORE, and Theorem 4.19, which bounds the step complexity of COLLECT. Most of our analysis is geared towards establishing the latter pair of theorems. We remark that the constant factors associated with our bounds could be improved via a more careful analysis. In general we have opted to simplify the presentation at the expense of such constant factors.

4.1 A Randomized Splitter

Algorithm 2 presents the code defining the operation of our randomized splitter. If only one process enters the splitter, it is guaranteed to stop. If two or more processes enter the splitter, then zero or one processes stop, and the remaining processes each get a return value of **left** or **right**, independently and uniformly at random.

4.2 Randomized Splitter Trees

A *randomized splitter tree* is a complete binary tree with a randomized splitter at each vertex. A process enters a randomized splitter tree at the root and attempts to acquire the root vertex by entering the associated randomized splitter. If this attempt is successful, the process stops at the root randomized splitter. Otherwise, the process recursively descends to one of the two subtrees of the root depending on the value, left or right, returned by the root randomized splitter. A process is said to stop in the tree if it successfully acquires some vertex. A vertex that is visited by at least one process is said to be *marked*.

Randomized splitter trees are the basic building block of the randomized adaptive collect algorithm to be presented in Section 4.3. In this section, we establish a number of basic probabilistic lemmas characterizing the behavior of this building block. Throughout the remainder of Section 4, we find it convenient to employ a shorthand notation to characterize the probability with which certain claims hold. In particular, when we say that a claim holds "whp(a)", where a is a parameter, we mean that the probability that the claim fails to hold is upper bounded by an arbitrary inverse polynomial in a. In other words, the claim holds with probability at least $1 - a^{-c}$, where c is a positive constant that can be set arbitrarily large by appropriately adjusting other constants in the relevant context.

A basic technical tool that we use is the following standard bound on the upper tail of the binomial distribution. Let X denote a random variable drawn from $B(n, p)$, that is, assume that X is the number of successes observed in n independent Bernoulli trials, each with success probability p. Then the following

inequality holds for all nonnegative δ:

$$\Pr(X \geq (1+\delta)np) \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{np} \tag{3}$$

At times it will be convenient to use the following weakened version of the preceding inequality, which holds for all $\alpha \geq 1$. This version may be derived from Equation (3) by observing that $e^{\delta} < e^{1+\delta}$ and setting $\alpha = \delta + 1$.

$$\Pr(X \geq \alpha np) \leq \left(\frac{e}{\alpha}\right)^{\alpha np} \tag{4}$$

We also make use of the following bound on the lower tail of the binomial distribution, which holds for all δ in the interval $[0, 1]$.

$$\Pr(X \leq (1-\delta)np) \leq \exp(-\delta^2 np/2) \tag{5}$$

See [5] or [17], for example, for derivations of Equations (3) and (5).

For any pair of real-valued random variables X and Y, we say that X *dominates* Y if for all reals z, $\Pr(X \geq z) \geq \Pr(Y \geq z)$. The following sequence of lemmas are concerned with the random experiment in which b processes enter a randomized splitter tree with a leaves, where a and b are positive integers such that $b \leq a$.

Lemma 4.1. *The number of processes leaving the tree is dominated by a random variable drawn from* $2B(b, b/a)$.

Proof. Fix an arbitrary numbering of the processes from 1 to b. For any process x, let $E_0(x)$ denote the event that x leaves the tree, let $E_1(x)$ denote the event that x descends to a leaf and at least one other process descends to the same leaf, and let $E_2(x)$ denote the event that x descends to a leaf and at least one other lower-numbered process descends to the same leaf. Let the random variable X (resp., Y, Z) denote the total number of processes x such that event $E_0(x)$ (resp., $E_1(x)$, $E_2(x)$) occurs.

For any leaf v, let the random variable $W(v)$ denote the number of processes that descend to leaf v. Note that $Y = \sum_{v:W(v)>1} W(v)$ while $Z = \sum_{v:W(v)>1} W(v) - 1$. It follows that $2Z \geq Y$, so the random variable $2Z$ dominates Y. Furthermore, Y dominates X since event $E_1(x)$ occurs whenever event $E_0(x)$ occurs. Thus $2Z$ dominates X, and we can complete the proof of the lemma by showing that Z is dominated by a random variable drawn from $B(b, b/a)$.

To see that Z is dominated by a random variable drawn from $B(b, b/a)$, consider a modified version of the random experiment in which no process stops or fails at an internal vertex of the randomized splitter tree, i.e., each process descends randomly from the root until it reaches a leaf. For this modified random experiment, let $E_2'(x)$ denote the event that a process x descends to the same leaf as at least one other lower-numbered process. Let the random variable Z' denote the total number of processes x such that $E_2'(x)$ occurs. Note that we can convert a run of the original experiment to a run of the modified experiment as follows: For each process x that stops or fails at some internal vertex v in the original experiment, randomly extend the path of x downward from v to a leaf. Observe that in such a pair of runs of the original and modified experiment, for any process x, if $E_2(x)$ occurs in the original experiment then $E_2'(x)$ occurs in the modified experiment. It follows that Z' dominates Z.

We now complete the proof by showing that Z' is dominated by a random variable drawn from $B(b, b/a)$. One way to run the modified experiment is to consider the processes one at a time in numerical order, and to generate a uniformly random root-leaf path for each process. Running the experiment in this manner, we see that Z' counts the number of times a process selects a previously selected path. Since the probability any process selects a previously selected path is at most $(b-1)/a < b/a$, Z' is dominated by a random variable drawn from $B(b, b/a)$. □

Lemma 4.2. *The number of processes that leave the tree is upper bounded by* $\max(4b^2/a, \mathrm{O}(\log a))$ *whp*(a).

Proof. By Lemma 4.1, it is sufficient to prove that if X is a random variable drawn from $B(b, b/a)$, then X is at most $\max(2b^2/a, \mathrm{O}(\log a))$ whp(a). In other words, we wish to prove that the probability X exceeds $\max(4b^2/a, c\log a)$ can be driven below an arbitrary inverse polynomial in a by making a sufficiently large choice of the positive constant c.

To see this, let us first assume that $c\log a \leq 8b^2/a$ and consider Equation (3) with $n = b$, $p = b/a$, and $\delta = 1$. With this choice of the parameters, Equation (3) implies that the probability X exceeds $2b^2/a$ is at most $(e/4)^{b^2/a} \leq (e/4)^{(c/8)\log a} = a^{-c'}$ where $c' = (2 - \log_2 e)c/8 \approx 0.06966c$. Thus this probability can be made smaller than an arbitrary inverse polynomial in a by choosing the constant c sufficiently large.

Now let us assume that $c\log a \geq 8b^2/a$. In this case, consider Equation (4) with $n = b$, $p = b/a$, and $\alpha = \frac{ac\log a}{b^2}$ so that $\alpha np = c\log a$. With this choice of parameters, Equation (4) implies that the probability X exceeds $c\log a$ is at most $(e/\alpha)^{c\log a}$. Now observe that $\alpha \geq 8 \geq 2e$ since $c\log a \geq 8b^2/a$. Thus the probability that $X \geq c\log a$ is at most $2^{-c\log a} = a^{-c}$, completing the proof. □

Lemma 4.3. *If* $b = \mathrm{O}(a^{1/3})$*, then the number of processes that leave the tree is* $\mathrm{O}(1)$ *whp*(a).

Proof. By Lemma 4.1, it is sufficient to prove that if X is a random variable drawn from $B(b, b/a)$ and $b = \mathrm{O}(a^{1/3})$, then the probability that X exceeds a sufficiently large positive constant is less than an arbitrary inverse polynomial in a.

To see this, consider Equation (4) with $n = b$, $p = b/a$, and $\alpha = \frac{ac}{b^2}$ for some positive constant c. With this choice of parameters, Equation (4) implies that the probability X exceeds c is at most

$$\left(\frac{b^2 e}{ac}\right)^c \quad = \quad \mathrm{O}(a^{-c/3}),$$

where the preceding equation follows from our assumption that $b = \mathrm{O}(a^{1/3})$. This probability can be driven below an arbitrary inverse polynomial in a by making a sufficiently large choice of the positive constant c. □

Lemma 4.4. *If* $b = \mathrm{O}(1)$ *then the probability that no processes leave the tree is* $1 - \mathrm{O}(1/a)$.

Proof. By Lemma 4.1, it is sufficient to prove that if X is a random variable drawn from $B(b, b/a)$ and $b = \mathrm{O}(1)$, then the probability that $X \geq 1$ is $\mathrm{O}(1/a)$.

To see this, consider Equation (4) with $n = b$, $p = b/a$, and $\alpha = \frac{a}{b^2}$. With this choice of parameters, Equation (4) implies that the probability $X \geq 1$ is at most $b^2 e/a$, which is $\mathrm{O}(1/a)$ for $b = \mathrm{O}(1)$. □

Lemma 4.5. *Let* X *denote a random variable equal to the number of independent flips of a fair coin required to obtain* $b - 1$ *heads. Then the number of marked vertices is dominated by* $X + b$*.*

Proof. Call a marked vertex *good* if some process stops or fails at the vertex, and *bad* otherwise. Note that there are at most b good vertices. Below we complete the proof of the lemma by arguing that the number of bad vertices is dominated by X.

Note that two or more processes leave each bad vertex. Call a bad vertex v *unlucky* if all of the processes leaving v descend to the same child of v. Call a bad vertex *lucky* otherwise.

We claim that at most $b - 1$ bad vertices are lucky. One way to see this is to reveal the downward paths of all processes in a breadth-first manner starting at the root. While doing this, we maintain a partition of

the processes into equivalence classes based on the portions of their paths that have been revealed thus far. Initially, all processes belong to a single equivalence class since all of their associated paths are empty. When a lucky bad vertex v is encountered, the equivalence class of processes descending to v is partitioned into two or three nonempty equivalence classes. (A three-way partition is possible because one process could stop at v.) Suppose we encounter a $(b-1)$th lucky bad vertex. Then at that point we have exactly b singleton equivalence classes, and so we cannot encounter another bad vertex. To complete the proof, note that each bad vertex we encounter has probability at most $1/2$ of being unlucky, independent of the luckiness of any previously identified bad vertices. It follows that the number of bad vertices is dominated by X. □

Lemma 4.6. *The expected number of marked vertices is* $\mathrm{O}(b)$.

Proof. Immediate from Lemma 4.5. □

Lemma 4.7. *The number of marked vertices is* $\mathrm{O}(b)$ *whp*(b).

Proof. Let the random variable X be as defined in the statement of Lemma 4.5. By Lemma 4.5, it is sufficient to prove that $X = \mathrm{O}(b)$ whp(b). Let Y denote the number of heads in $4b$ flips of a fair coin. In order to establish the desired bound on X, it is sufficient to prove that $Y \geq b$ whp(b). The latter claim is immediate from Equation (5) with $n = 4b$, $p = 1/2$, and $\delta = 1/2$. Remark: The inverse polynomial bound on the failure probability claimed in this lemma is somewhat weaker than what is implied by Equation (5), but is adequate for our purposes. □

Lemma 4.8. *The number of marked vertices is* $\mathrm{O}(b + \log a)$ *whp*(a).

Proof. Let the random variable X be as defined in the statement of Lemma 4.5. By Lemma 4.5, it is sufficient to prove that $X = \mathrm{O}(b + \log a)$ whp(a). Let c be a positive integer constant, and let Y denote the number of heads in $4b + c \log a$ flips of a fair coin. Letting $n = 4b + c \log a$, $p = 1/2$, and $\delta = 1/2$ in Equation (5), we find that the probability $Y \leq b + \frac{c \log a}{4}$ is at most $\exp(-\frac{b}{4} - \frac{c \log a}{16}) \leq \exp(-\frac{c \log a}{16})$. It follows that the number of flips required to obtain $b - 1$ heads is at most $4b + \mathrm{O}(\log a) = \mathrm{O}(b + \log a)$ whp(a). □

Lemma 4.9. *The maximum depth of any marked vertex is* $\mathrm{O}(\log b)$*, both whp*(b) *and expected.*

Proof. The probability that two processes follow the same downward path to depth i is at most 2^{-i}. By a union bound, the probability that any pair of the b processes follow the same downward path to depth i is $q_i = \mathrm{O}(b^2 2^{-i})$. The whp$(b)$ claim follows since a vertex at level $i + 1$ can only be marked if two or more processes follow the same downward path to depth i. The bound on the expectation follows since the q_i's decrease geometrically with i. □

4.3 The Construction

Our randomized adaptive collect algorithm employs a cascaded sequence of randomized splitter trees T_i, $1 \leq i \leq \ell$, where $\ell = \mathrm{O}(\log \log n)$, along with a backup array of size n. (See Figure 3.) Assume without loss of generality that n is a power of 2. Then tree T_i has $n_i = n \cdot 2^{5-i}$ leaves and its depth is $\log n + 5 - i$. As in Figure 2, for each tree T_i such that $i < \ell$, both children of all the leaves of T_i are defined to be the root of T_{i+1}. Both children of all the leaves of T_ℓ are defined to be **nil**. On the first invocation of a STORE operation, a process enters T_1 and proceeds downward as described in Section 4.2 until it either stops at a vertex of some T_i — thereby successfully acquiring the register associated with that vertex — or leaves T_ℓ. In the latter case, the process raises a global flag (called *overflow*) to indicate that the backup array is in

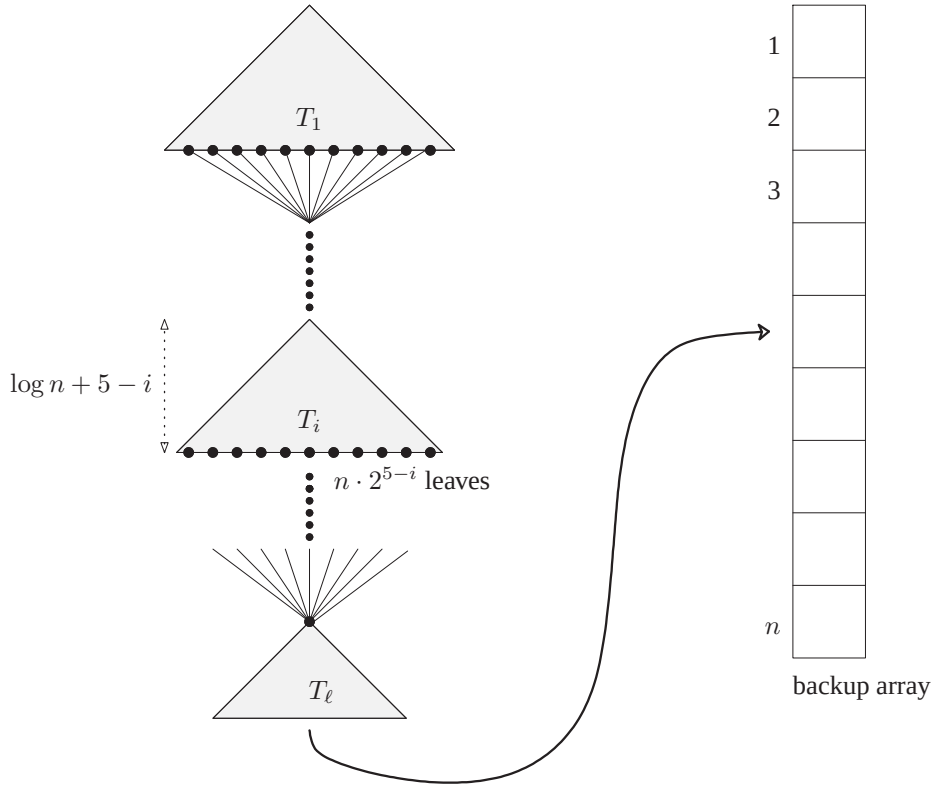

Figure 3: Cascaded randomized splitters trees.

use, and acquires the array register corresponding to its ID. That is, process i acquires register i of the array, where $1 \leq i \leq n$. In either case, the process completes the STORE operation by writing the value into the acquired register. Subsequent STORE operations by the same process are completed in a constant number of operations by writing into the register acquired previously. Of course, a process may fail at any point during its execution.

The code for acquiring a vertex is similar to Algorithm 1, and appears in Algorithm 3.

The COLLECT works analogously to the previous algorithms. The marked vertices of T_1 are traversed in DFS order. Then, if the root of T_2 is marked, the marked vertices of T_2 are traversed, and so on. Finally, if the flag of the array (*overflow*) is set, the entire backup array is read.

4.4 Analysis

We now analyze the performance of our adaptive collect algorithm in terms of the parameters n and k. The memory complexity of the algorithm is straightforward to analyze.

Theorem 4.10. *The memory complexity is* $\mathrm{O}(n)$.

Proof. The T_i's are geometrically decreasing in size, and T_1 has size $\Theta(n)$, so the total size of all the T_i's is linear in n. The size of the backup array is also linear in n. □

Algorithm 3 Cascaded randomized splitter trees: Node acquisition

```
1: v = root of T_1
2: repeat
3:    v.mark = true
4:    move = rand-splitter(v) {returns either stop, left, or right}
5:    if move == left then
6:       v = v.left-child
7:    else if move == right then
8:       v = v.right-child
9:    fi
10: until move == stop or v == nil
11: if move == stop then
12:    v.id = id {write your identifier}
13:    return(v)
14: fi
15: overflow = true {the backup array is used}
16: return(backup[id])
```

Our remaining goal is to bound the step complexity of the STORE and COLLECT operations. To this end, we first present a few auxiliary definitions and lemmas.

For all i such that $1 \leq i \leq \ell$, let k_i denote the number of processes entering T_i. Thus $k = k_1$. In addition, it is convenient to define $k_{\ell+1}$ as the number of processes entering the backup array.

Throughout the remainder of this section, let c denote a sufficiently large positive constant. Call a tree *marked* if at least one of its vertices is marked, that is, T_i is marked if and only if $k_i > 0$. Assign a color to each tree T_i as follows. Each unmarked tree is white. If there is no marked tree T_i such that $k_i \leq c \log n$, then all marked trees are red. Otherwise, the marked tree T_i with the least index i such that $k_i \leq c \log n$ is purple, all (marked) trees with lower indices are red, and all marked trees with higher indices are blue.

In several of the proofs that follow, we make implicit use of the fact that any claim holding whp(n_i), where $1 \leq i \leq \ell$, also holds whp(n). This is because n_i is within a polylogarithmic factor of n, and as such is lower-bounded by a polynomial in n.

Lemma 4.11. *If tree T_i is red, then $k_{i+1}/n_{i+1} \leq \max(8(k_i/n_i)^2, \mathrm{O}(\log n_i)/n_i)$ whp(n).*

Proof. Lemma 4.2 implies that $k_{i+1} \leq \max(4k_i^2/n_i, \mathrm{O}(\log n_i))$ whp(n_i), and hence whp(n). The claim follows by dividing through by n_i and using the fact that $n_{i+1} = n_i/2$. □

It is convenient to define the following function for all positive integers a and b such that $a \geq b$.

$$f(a, b) = \max[1, (\log \log a) - \log \log(2a/b)] \tag{6}$$

Note that $f(a, b) = \mathrm{O}(\log \log b)$, and $f(a, b) = \mathrm{O}(1)$ for $b \leq a^{1-\varepsilon}$, where ε denotes an arbitrarily small positive constant.

Lemma 4.12. *There are $\mathrm{O}(f(n, k))$ red trees whp(n).*

Proof. If $k_1 \leq c \log n$, then there are no red trees, so the claim is trivial. In what follows, we assume that $k_1 \geq c \log n$. By Lemma 4.11, whp(n), either k_2 is $\mathrm{O}(\log n)$ or k_2/n_2 is at most $8(k_1/n_1)^2$. If k_2

is $\mathrm{O}(\log n)$, then assuming we choose the positive constant c sufficiently large, there is exactly one red tree. Otherwise, there are at least two red trees. So we may assume in what follows that k_2/n_2 is at most $8(k_1/n_1)^2$. By Lemma 4.11, whp(n), either k_3 is $\mathrm{O}(\log n)$ or k_3/n_3 is at most $8(k_2/n_2)^2 \leq 8^3(k_1/n_1)^4$. If k_3 is $\mathrm{O}(\log n)$, then assuming we choose the positive constant c sufficiently large, there are exactly two red trees. Otherwise, there are at least three red trees. So we may assume in what follows that k_3/n_3 is at most $8^3(k_1/n_1)^4$. Continuing in this manner, we find that after i iterations, either we have exhausted all of the red trees, or

$$\begin{aligned} k_i/n_i &\leq 8^{2^i-1}(k_1/n_1)^{2^i} \\ &= \frac{1}{8}(8k_1/n_1)^{2^i}. \end{aligned}$$

Thus we can obtain a whp(n) upper bound the number of red trees by determining the maximum i such that the preceding upper bound on k_i/n_i is at least $1/n$, say, since $1/n$ is $\mathrm{O}(\frac{\log n}{n_i})$ for any i. Taking logarithms, and using $k = k_1$ and $n_1 = 16n$, we seek the maximum i such that $2^i \log \frac{k}{2n} \geq \log \frac{8}{n}$, or equivalently, $2^i \log \frac{2n}{k} \leq \log \frac{n}{8}$. Taking logarithms once again, and rearranging terms, we find that whp(n) the number of red trees is at most $\log\log \frac{n}{8} - \log\log \frac{2n}{k} \leq f(n,k)$. □

Lemma 4.13. *If T_i is blue then $k_i = \mathrm{O}(1)$ whp(n).*

Proof. If T_i is blue then there is a purple tree T_j such that $j < i$. Lemma 4.3 implies that k_{j+1} is $O(1)$ whp(n). The claim follows since $k_i \leq k_{j+1}$. □

Lemma 4.14. *There are $\mathrm{O}(1)$ blue trees whp(n).*

Proof. Assume that there are one or more blue trees and let T_i be the blue tree with the least index. By Lemma 4.13, $k_i = \mathrm{O}(1)$ whp(n). By repeated application of Lemma 4.4 we find that, conditional on $k_i = \mathrm{O}(1)$, the probability that there are more than m blue trees is $O(a^{-m})$ for any positive constant m. The claim of the lemma follows. □

Lemma 4.15. *There are $\mathrm{O}(f(n,k))$ marked trees whp(n).*

Proof. Recall that every marked tree is either red, purple, or blue, and there is at most one purple tree. Thus the claim follows from Lemmas 4.12 and 4.14 and the observation that $f(n,k) \geq 1$. □

Lemma 4.16. *We can choose ℓ such that $\ell = \mathrm{O}(\log\log n)$ and whp(n) the backup array is unused.*

Proof. It is sufficient to prove that the total number of marked trees is $\mathrm{O}(\log\log n)$ whp(n). This is immediate from Lemma 4.15 since $f(n,k) = \mathrm{O}(\log\log n)$ for all k. □

Lemma 4.17. *The expected number of marked trees is $\mathrm{O}(f(n,k))$.*

Proof. By our choice of ℓ, the maximum number of marked trees is $\mathrm{O}(\log\log n)$. Thus the desired bound on the expected number of marked trees follows from Lemma 4.15. □

We are now ready to state and prove the two main theorems of this section.

Theorem 4.18. *The step complexity of the first invocation of* STORE *satisfies the following upper bounds: $\mathrm{O}((\log n)\log\log n)$ worst case; $\mathrm{O}(f(n,k)\log n)$ whp(n); $\mathrm{O}(f(n,k)\log k)$ whp(k); $\mathrm{O}(f(n,k)\log k)$ expected.*

Proof. Let the random variable X denote the the maximum depth of any marked node in the overall cascaded tree structure. Note that in order to establish the step complexity bounds claimed in the lemma, it is sufficient to establish that these bounds hold for the random variable X.

Note that each tree has $\mathrm{O}(\log n)$ depth. The worst case bound follows since there are $\ell = \mathrm{O}(\log \log n)$ trees. The whp(n) bound follows from Lemma 4.15. For the two remaining bounds, let us consider the cases $k \leq \sqrt{n}$ and $k \geq \sqrt{n}$ separately.

First assume that $k \leq \sqrt{n}$. At most k processes enter any marked tree, so Lemma 4.9 implies that whp(k) the maximum depth of any marked vertex within a marked tree T_i (i.e., relative to the root of T_i) is $\mathrm{O}(\log k)$. Furthermore, $f(n,k) = \mathrm{O}(1)$ for $k \leq \sqrt{n}$, so the number of marked trees is $\mathrm{O}(1)$ whp(n) by Lemma 4.15. We conclude via a union bound that whp(k) there are $\mathrm{O}(1)$ marked trees and that all marked vertices in any marked tree T_i occur at depth $\mathrm{O}(\log k)$ within T_i. It follows that X is $\mathrm{O}(\log k)$ whp(k), as required. To bound the expected value of X, note that Lemma 4.17 implies that the expected number of marked trees is $\mathrm{O}(1)$. Furthermore, since at most k processes enter any marked tree, Lemma 4.9 implies that the expected maximum depth of any marked node within any marked tree is $\mathrm{O}(\log k)$. It follows that the expected value of X is $\mathrm{O}(\log k)$.

Now assume that $k \geq \sqrt{n}$. In this case, $\log k = \Omega(\log n)$, so it is sufficient to establish a bound of $\mathrm{O}(f(n,k) \log n)$, both whp$(k)$ and expected. But both of these bounds are immediate from our whp(n) bound. □

Theorem 4.19. *The step complexity of* COLLECT *satisfies the following upper bounds:* $\mathrm{O}(n)$ *worst case;* $\mathrm{O}(k + \log n)$ *whp*(n)*;* $\mathrm{O}(k)$ *whp*(k)*;* $\mathrm{O}(k)$ *expected.*

Proof. Let the random variable X denote the total number of marked vertices in the randomized splitter trees. Lemma 4.16 implies that in order to establish the step complexity bounds claimed in the lemma, it is sufficient to establish that these bounds hold for the random variable X.

The $\mathrm{O}(n)$ worst case bound on X is immediate from Theorem 4.10.

Lemma 4.2 implies that the sequence of k_i's associated with the red trees decreases (super-)geometrically whp(n). Thus, Lemma 4.8 implies that the number of marked vertices in all red trees is $\mathrm{O}(k)$ whp(n). The number of marked vertices in the purple tree, if any, is $\mathrm{O}(\log n)$ whp(n) by Lemma 4.8. The number of marked vertices in all blue trees is $\mathrm{O}(\log n)$ whp(n) by Lemmas 4.13 and 4.14, and the fact that the depth of every tree is $\mathrm{O}(\log n)$. Thus $X = \mathrm{O}(k + \log n)$ whp(n).

Now let us prove that $X = \mathrm{O}(k)$ whp(k). If T_1 is red, then $k = \Omega(\log n)$, so we have $X = \mathrm{O}(k + \log n) = \mathrm{O}(k)$ whp(n), implying that $X = \mathrm{O}(k)$ whp(k). Otherwise, T_1 is purple, and Lemma 4.7 implies that the number of marked vertices in T_1 is $\mathrm{O}(k)$ whp(k). Furthermore, Lemma 4.9 implies that whp(k) there are no blue trees, so $X = \mathrm{O}(k)$ whp(k).

It remains to prove that the expectation of X is $\mathrm{O}(k)$. If T_1 is red, then $k = \Omega(\log n)$ so $X = \mathrm{O}(k + \log n) = \mathrm{O}(k)$ whp(n). The latter bound implies that the expectation of X is $\mathrm{O}(k)$, since $X = \mathrm{O}(n)$ in the worst case. If T_1 is not red then it is purple, so $k = \mathrm{O}(\log n)$ and Lemma 4.17 implies that the expected number of marked trees is $\mathrm{O}(1)$. By the expectation bound of Lemma 4.6, the expected value of X is $\mathrm{O}(k)$ times the expected number of marked trees, and hence is $\mathrm{O}(k)$. □

5 Conclusions

We presented new deterministic and randomized adaptive collect algorithms. Table 1 compares the algorithms presented in this paper with previous work. The algorithms are adaptive to so-called *total contention*, that is, to the maximum number of processes that were ever active during the execution. There are other

Algorithm	Step Complexity		Memory Complexity	
	COLLECT	STORE		
triangular matrix [16]	$\mathrm{O}(k^2)$	$\mathrm{O}(k)$	$\mathrm{O}(n^2)$	deterministic
tree [9]	$\mathrm{O}(k)$	$\mathrm{O}(k)$	$\mathrm{O}(2^n)$	deterministic
cascaded trees (Sec. 3.2)	$\mathrm{O}(k^2/(\varepsilon \log n))$	$\mathrm{O}(k/\varepsilon)$	$\mathrm{O}(n^{2+\varepsilon})$	deterministic
randomized splitters (Sec. 4)	$\mathrm{O}(k)$	$\mathrm{O}(f(n,k)\log k)$	$\mathrm{O}(n)$	randomized

Table 1: Summary of the complexities achieved by different collect algorithms. See Equation (6) for the definition of the function f.

contention definitions which are more fine-grained, such as point contention. The *point contention* during an execution interval is the maximum number of processes that were simultaneously active at some point in time during that interval. We believe that some of our new techniques carry over to algorithms that adapt to point contention [2, 3, 8].

Our paper shows that it is possible to perform a COLLECT operation in $\mathrm{O}(k)$ time with polynomial memory using randomization. To determine the best possible step complexity for COLLECT achievable by a deterministic algorithm with polynomial memory is an interesting open problem.

References

[1] Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 105–112, 1999.

[2] Y. Afek, G. Strupp, and D. Touitou. Long-lived adaptive collect with applications. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 262–272, 1999.

[3] Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive atomic snap-shot and immediate snapshot. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 71–80, 2000.

[4] Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[5] N. Alon and J. H. Spencer. *The Probabilistic Method*. Wiley, New York, NY, 1991.

[6] J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:75–110, 2003.

[7] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. *Distributed Computing*, 15(3):177–189, 2002.

[8] H. Attiya and A. Fouren. Algorithms adaptive to point contention. *Journal of the ACM*, 50(4):444–468, July 2003.

[9] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

[10] H. Attiya, F. Kuhn, M. Wattenhofer, and R. Wattenhofer. Efficient adaptive collect using randomization. In *Proceedings of the 18th Annual Conference on Distributed Computing*, volume 3274 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2004.

[11] R. Guerraoui. Indulgent algorithms. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, number 289–297, 2000.

[12] R. Guerraoui and M. Raynal. A generic framework for indulgent consensus. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 88–95, 2003.

[13] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[14] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

[15] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 14th International Symposium on Distributed Computing*, volume 2180 of *Lecture Notes in Computer Science*, pages 1–15, 2001.

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

[17] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, UK, 1995.