Efficient Adaptive Collect using Randomization

Hagit Attiya[1], Fabian Kuhn[2], Mirjam Wattenhofer[2], and Roger Wattenhofer[2]

[1] Department of Computer Science, Technion
[2] Department of Computer Science, ETH Zurich

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 polynomial memory overhead only. In addition we present a new deterministic collect algorithm which 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 [9, 10], 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, 4, 8], mutual exclusion [2, 3, 5, 6], 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 $f(k)$ steps when there are k participating processes, we say that it is f-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 [11].

Several adaptive, wait-free collect algorithms are known [2, 7, 8]. In particular, there is an algorithm that features an asymptotically optimal $\mathrm{O}(k)$-adaptive collect, but its memory consumption is exponential in the number of potential processes [8], 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 [8, 14].[3]

[3] Moir and Anderson [14] employ a matrix structure to solve the renaming problem. The same structure can be used to solve the collect problem, following ideas of [8].

The lower bound of Jayanti, Tan and Toueg [12] 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 practical, in contrast to known deterministic algorithms with either super-linear step complexity or unrealistic memory overhead. We present wait-free algorithms that take $\mathrm{O}(k)$ steps to store and collect, while having polynomial memory overhead only. The algorithms are randomized, and their 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, both randomized algorithms use *splitters* as introduced by Moir and Anderson to govern the algorithmic decisions of processes [14]. Our first algorithm (Section 4) uses a *randomized* splitter, and operates on a complete binary tree of depth $c \log n$, for carefully chosen constant c. A process traverses the tree of random splitters as in the linear collect algorithm [8]. We prove that with high probability the process stops at some vertex in this shallow tree; in (the low-probability) case that a process reaches the leaves of the tree, it falls back on a deterministic *backup* structure. A binary tree of radomized splitters was previously used by Kim and Anderson [13] for adaptive mutual exclusion.

Our second algorithm (Section 5) uses standard, deterministic splitters [14]. The splitters are connected in a random graph (with out-degree two), that is, the randomization is in the topology rather than in the actual algorithm executed by the processes. A process traverses the random graph by accessing the splitters. However, if the process suspects that it has stayed in the graph for too long, it immediately moves to a deterministic backup structure. We prove that with high probability, the graph traversed by the processes does not contain a cycle, and the backup structure is not accessed at all. This relies on the assumption that the adversarial scheduler is not allowed to inspect this graph.

The crux of the step complexity analysis of both algorithms is a balls-into-bins game, and it requires a probabilistic lemma estimating the number of balls in bins containing more than one ball.

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. The randomized algorithms fall back on this algorithm. 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 [8, 14].

All new algorithms build on the basic collect algorithm on a binary tree [8]. 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* during α if α includes a call of a procedure F_i. The *total contention* during α is the number of all processes that are active during α. Let f be a non-decreasing function. An implementation is f*-adaptive* to total contention if the step complexity of each of its procedures is bounded from above by $f(k)$, where k is the total contention.

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* [14]: 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 [8], 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 **right**, it moves to the right child. A process marks each vertex it accesses by raising a flag associated with the vertex. 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 later invocations of STORE, process i immediately writes its value in $v.value$, clearly leading to a constant step complexity $\mathrm{O}(1)$. 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 1. *The memory complexity is* $\Theta(2^n)$.

Lemma 2 ([8]). *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. *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:

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

depending on whether (1) or not (2) a process stops in the splitter.

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):

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

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.

The Algorithm The algorithm is performed on a sequence of $n/((\gamma-1)\lceil\log n\rceil)$ complete binary splitter trees of depth $\gamma \log n$, denoted $T_1, \ldots, T_{n/((\gamma-1)\lceil\log n\rceil)}$. Except for the last tree, each leaf of tree T_i has an edge to the root of tree T_{i+1} (Figure 1).

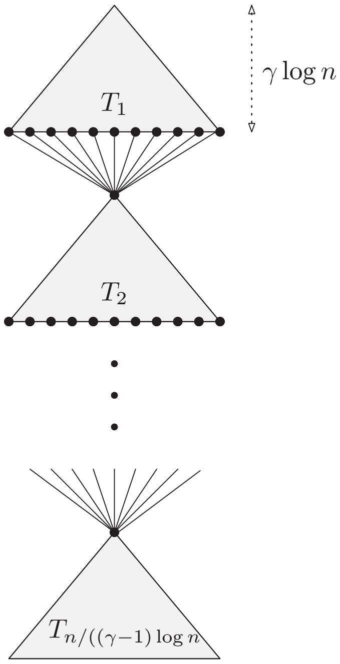

Fig. 1. Organization of splitters in the cascaded trees algorithm.

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 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.

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}.

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

Lemma 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 5. *At least* $\min\{k_i, (\gamma-1)\lceil\log n\rceil\}$ *processes stop in some vertex of tree* T_i*, for every* i*,* $1 \leq i \leq n/(\gamma-1)\log n$*.*

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

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 of v is visited in T_i, then some process obtained **stop** at v and does not continue. Otherwise, processes reaching v are split between 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 left 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 at each such node, the number of paths to the leaves increases by one.

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

Starting from the leaves, the number of vertices on each preceding level is at least half the number at the current level. For the number of non-leaf vertices n_i of tree T_i', we therefore get:

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

where the number of ones in the equation follows from the fact that the tree T_i hast depth $\gamma\log n$ and after $\lceil\log m_i\rceil$ levels the number of vertices on the next level can be lower bounded by one. The claim follows since $m_i \leq n$. $\square$

Lemma 6. *A process writes its id in a vertex at depth at most* $k \cdot \gamma/(\gamma-1)$*.*

Proof. If $k \leq (\gamma-1)\lceil\log n\rceil$, the claim follows from Lemma 2.

If $k = m \cdot (\gamma-1)\lceil\log n\rceil$, for some $m > 1$, then we know by Lemma 5 that in each tree, at least $(\gamma-1)\lceil\log n\rceil$ processes will stop in a vertex. Thus, at most $(\gamma-1)\lceil\log n\rceil$ processes access tree T_m. By Lemma 2, a process stops in a vertex with total depth at most $\gamma\log n \cdot (m-1) + \gamma\log n = k \cdot \gamma/(\gamma-1)$. $\square$

Since each splitter requires a constant number of operations, by Lemma 6, the step complexity of the first invocation of STORE is $\mathrm{O}(\gamma/(\gamma-1)k)$ and all invocations thereafter require $\mathrm{O}(1)$ steps.

By Lemma 3, the time to collect in tree T_i is $2k_i - 1$. By Lemma 6, all processes stop after at most $k/((\gamma-1)\log n)$ trees. This implies the next lemma:

Lemma 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 [14]. 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 [8].

4 Adaptive Collect with Randomized Splitters

The next two sections present two algorithms that allow to STORE and COLLECT with $\mathrm{O}(k)$ step complexity and polynomial memory complexity. For the first algorithm, we use a new kind of randomized splitters, arranged in a binary tree of small size. The second algorithm uses classical splitters (Section 3.1) which are interconnected at random.

For both algorithms, we need the following lemma.

Lemma 8. *Assume k balls are thrown into N bins, i.e. the bins for all balls are chosen independently and uniformly at random. Let C denote the number of balls ending up in bins containing more than one ball. We have*

$$\frac{k(k-1)}{N} - \frac{k^3}{2N^2} \;\leq\; \mathrm{E}[C] \;\leq\; \frac{k(k-1)}{N} \tag{3}$$

and

$$\Pr\left(C \geq t + \frac{k^2}{N}\right) \;\leq\; \frac{6k^2}{t^2 N} \tag{4}$$

under the condition that $k \leq N^{2/3}$.

Proof. Let us first prove the bounds (3) on the expected value $\mathrm{E}[C]$. The random variable B_m denotes the number of bins containing exactly m balls. Further, P is the number of pairs (i,j) of balls for which ball i and ball j end up in the same bin. The variable T is defined accordingly for triples. Clearly, we have

$$C = \sum_{m=2}^{\infty} m B_m,$$

as well as

$$P = \sum_{m=2}^{\infty} \binom{m}{2} B_m \quad \text{and} \quad T = \sum_{m=3}^{\infty} \binom{m}{3} B_m.$$

We get $2P - 3T \le C \le 2P$ because

$$\begin{aligned}2P - 3T &= 2B_2 + \sum_{m=3}^{\infty} m(m-1)\left(1 - \frac{m-2}{2}\right) B_m \\ &\le 2B_2 + 3B_3 \ \le\ C \le \sum_{m=2}^{\infty} m(m-1)B_m \ =\ 2P.\end{aligned}$$

Let p_{ij} be the probability that a pair of balls i and j are in the same bin. Accordingly, p_{ijl} denotes the probability that balls i, j, and l are in the same bin. We have $p_{ij} = 1/N$ and $p_{ijl} = 1/N^2$. Using p_{ij} and p_{ijl}, we can compute the expected values of P and T as

$$\begin{aligned}\mathrm{E}[P] &= \binom{k}{2} p_{ij} = \frac{k(k-1)}{2N} \quad \text{and} \\ \mathrm{E}[T] &= \binom{k}{3} p_{ijl} \le \frac{k^3}{6N^2}.\end{aligned}$$

Using $2P - 3T \le C \le 2P$ and linearity of expectation, the bounds on $\mathrm{E}[C]$, claimed in (3), follow.

We prove Inequality (4) using the Chebyshev inequality. In order to do so, we need to know an upper bound on the variance $\mathrm{Var}[C]$ of C. Let $\mathcal{E}_i$ be the event that ball i is in a bin with at least two balls. Further, $\mathcal{E}_{ij}$ is the event that i and j are together in the same bin whereas $\overline{\mathcal{E}_{ij}}$ denotes the complement, i.e. the event that i and j are in different bins. $\mathrm{Var}[C]$ can be written as $\mathrm{Var}[C] = \mathrm{E}[C^2] - \mathrm{E}[C]^2$. The expected value of C^2 can be computed as follows:

$$\begin{aligned}\mathrm{E}\big[C^2\big] &= \mathrm{E}\left[\left(\sum_{i=1}^{k} X_i\right)^2\right] = \mathrm{E}\left[\sum_{i=1}^{k} X_i^2 + 2\sum_{1\le i<j\le k} X_i X_j\right] \\ &= \mathrm{E}[C] + 2 \cdot \sum_{1\le i<j\le k} \Pr(\mathcal{E}_i \cap \mathcal{E}_j). \qquad (5)\end{aligned}$$

We have $\Pr(\mathcal{E}_i \cap \mathcal{E}_j) = \Pr(\mathcal{E}_i) \cdot \Pr(\mathcal{E}_j | \mathcal{E}_i)$ and

$$\begin{aligned}\Pr(\mathcal{E}_j|\mathcal{E}_i) &= \Pr(\mathcal{E}_{ij}|\mathcal{E}_i) + \Pr(\mathcal{E}_j \cap \overline{\mathcal{E}_{ij}}|\mathcal{E}_i) \\ &= \frac{\Pr(\mathcal{E}_{ij})}{\Pr(\mathcal{E}_i)} + \Pr(\mathcal{E}_2 \cap \overline{\mathcal{E}_{12}}|\mathcal{E}_{13}) \qquad (6) \\ &\le \frac{\Pr(\mathcal{E}_{ij})}{\Pr(\mathcal{E}_i)} + \Pr\left(\bigcup_{\ell=4}^{k} \mathcal{E}_{2\ell} \,\middle|\, \mathcal{E}_{13}\right) \\ &= \frac{\Pr(\mathcal{E}_{ij})}{\Pr(\mathcal{E}_i)} + \Pr\left(\bigcup_{\ell=4}^{k} \mathcal{E}_{2\ell}\right). \qquad (7)\end{aligned}$$

For Equation (6), we assume that w.l.o.g., ball 1 is in the same bin as ball 3 and that ball 2 shares the bin with some ball i for $i = 4, \ldots, k$. Equation (7) holds because $\mathcal{E}_{13}$ and

$\mathcal{E}_{2\ell}$ are independent for $\ell \geq 4$. The probability that two balls i and j are in the same bin is $\Pr(\mathcal{E}_{ij}) = 1/N$. Using the bounds (3) on $\mathrm{E}[C]$ and linearity of expectation, we get the following bounds for the probability of E_i:

$$\frac{k-1}{N} - \frac{k^2}{2N^2} \;\leq\; \Pr(\mathcal{E}_i) \;\leq\; \frac{k-1}{N}. \tag{8}$$

Therefore, we have

$$\frac{\Pr(\mathcal{E}_{ij})}{\Pr(\mathcal{E}_i)} \;\leq\; \frac{\frac{1}{N}}{\frac{k-1}{N} - \frac{k^2}{2N^2}} \;\leq\; \frac{2}{k-1} \tag{9}$$

where the second inequality of (9) holds for $k \leq N^{2/3}$ and $N \geq 5$. The second term of Equation (7) can be bounded as

$$\Pr\left(\bigcup_{\ell=4}^{k} \mathcal{E}_{2\ell}\right) \;\leq\; \sum_{\ell=4}^{k} \Pr(\mathcal{E}_{2\ell}) \;=\; \frac{k-3}{N}. \tag{10}$$

Combining (7), (8), (9), and (10), we get

$$\Pr(\mathcal{E}_i \cap \mathcal{E}_j) \leq \frac{k-1}{N} \cdot \left(\frac{2}{k-1} + \frac{k-3}{N}\right) \leq \frac{2}{N} + \frac{k^2}{N^2}.$$

Applying Equation (5), we have

$$\mathrm{E}[C^2] \leq \frac{k(k-1)}{N} + 2\binom{k}{2}\left(\frac{2}{N} + \frac{k^2}{N^2}\right) \leq \frac{3k^2}{N} + \frac{k^4}{N^2}$$

and therefore

$$\mathrm{Var}[C] \leq \frac{3k^2}{N} + \frac{k^4}{N^2} - \left(\frac{k(k-1)}{N} - \frac{k^3}{2N^2}\right)^2 \leq \frac{3k^2}{N} + \frac{2k^3}{N^2} + \frac{k^5}{N^3} \underset{(k \leq N^{3/2})}{\leq} \frac{6k^2}{N}.$$

Using the bounds for $\mathrm{E}[C]$ and $\mathrm{Var}[C]$, we can now apply Chebyshev in order to bound the probability of the upper tail of C:

$$\Pr\left(C \geq t + \frac{k^2}{N}\right) \;\leq\; \frac{6k^2}{t^2 N}.$$

This concludes the proof. □

4.1 The Algorithm

The algorithm presented in this section provides STORE and COLLECT with $\mathrm{O}(k)$ step complexity and polynomial memory complexity. It uses a new kind of splitter that makes a random choice in order to partition the processes to left and right. The algorithm operates on a complete binary tree of depth $c \log n$ ($c \geq 3/2$) with randomized splitters (see Algorithm 1) in the vertices. The algorithm uses the cascaded trees structure (of Section 3.2) as well as an array of length n as deterministic backup structures.

Algorithm 1 Randomized Splitter

$X = id_i$
if Y then return randomly **right** or **left**
Y = true
if $(X == id_i)$ **then**
    return **stop**
else
    return randomly **right** or **left**
fi

The cascaded trees have height $2\log(4\sqrt{n})$ and there are $4\sqrt{n}/\log(4\sqrt{n})$ such trees. That is, we build the structure with the parameter $\gamma = 2$ but only for $4\sqrt{n}$ processes.

As in the previous algorithms, a process tries to acquire a vertex by moving down the data structure according to the values obtained from the randomized splitters. If a process does not stop at a leaf of the tree, it enters the cascaded trees backup structure. If a process also fails to stop at a vertex of the first backup structure (the cascaded trees), it raises a flag, indicating that the array structure is used, and stores its value in the array at the index corresponding to the process ID. That is, process i stores its value at position i in the array for $1 \leq i \leq n$.

The COLLECT works analogously to the previous algorithms. The marked part (visited splitters) of the randomized splitter tree is first traversed in DFS order. Then, if necessary, the first backup structure is traversed as described in Section 3.2. Finally, if the flag of the array is set, the whole array is read.

4.2 Analysis

Clearly, the tree of randomized splitters needs $\mathrm{O}(n^c)$ randomized splitters and therefore $\mathrm{O}(n^c)$ registers. By Lemma 4, the first backup structure requires $\mathrm{O}((\sqrt{n})^3/\log\sqrt{n}) = \mathrm{O}(n^{3/2}/\log n)$ registers; the array takes n additional registers, implying the following lemma.

Lemma 9. *The memory complexity is $\mathrm{O}(n^c)$ for $c > 3/2$.*

Lemma 10. *The probability that more than $4\sqrt{k}$ processes enter the first backup structure is at most k/n^c.*

Proof. In order to get an upper bound on the probability that at least a certain number of processes reach a leaf of the randomized splitter tree, we can assume that whenever at least two processes visit a splitter, none of them stops and all of them are randomly forwarded to the left or the right child. Assume that we extend the random walk of stopping processes until they reach a leaf. If we do this, the set of processes which stops in the tree corresponds to the set of processes which arrive at a leaf that is not reached by any other processes. On the other hand, all processes which are not alone when arriving at their leaf have to enter the backup structure. Because the leaves of the processes are chosen independently and uniformly at random, we can view it as a 'balls-into-bins' game and the lemma follows by applying Lemma 8 with $N = n^c$ and $t = 3\sqrt{k}$. Note that $k \leq (n^c)^{2/3}$, since $c \geq 3/2$. $\square$

Lemma 11. *The number of marked nodes in the random splitter tree is at most $3k$ in expectation and no more than $7k$ with probability at least $1 - 1/2^k$.*

Proof. We partition the marked vertices into vertices where the processes are split (there are processes going left *and* processes going right *or* some process stops at the node) and into vertices where all processes proceed into the same direction. If we contract all the paths of vertices which do not split the processes to single edges, we obtain a binary tree where all vertices behave like regular splitters (not all processes go into the same direction). Hence, by Lemma 3, there are at most $2k - 1$ of those vertices. At most $k - 1$ of them are visited by more than one process. All paths of non-splitting vertices are preceding one of those $k - 1$ splitting vertices. That is, there are at most $k - 1$ paths of consecutive vertices where all processes proceed in the same direction. As there are at least two processes traversing such paths, in the worst case, the length Z_i of each path i is geometrically distributed with probability $\Pr(Z_i = \ell) \leq 1/2^{\ell+1}$. Thus, the distribution of the total number X of vertices where processes are not split can be by estimated by the distribution of the sum of $k - 1$ independent geometrically distributed random variables. Let $Y := \sum_{i=1}^{k-1} Z_i$, we have

$$\mathrm{E}[X] \;\leq\; \mathrm{E}[Y] \;=\; (k-1)\mathrm{E}[Z_i] \;=\; k-1.$$

and therefore, the total number of marked nodes is at most $3k$ in expectation. For the tail probability, we have

$$\Pr(X \geq x) \;\leq \Pr(Y \geq x). \tag{11}$$

The random variable Y can be seen as the number of Bernoulli trials with success probability $1/2$ until there are $k - 1$ successes, i.e., the distribution of Y is a negative binomial distribution. We have

$$\Pr(Y = y) \;=\; \binom{y-1}{k-2}\left(\frac{1}{2}\right)^y.$$

For $y \geq 5k$, we have

$$\frac{\Pr(Y = y+1)}{\Pr(Y = y)} \;=\; \frac{y}{y-k+2}\cdot\frac{1}{2} \;\leq\; \frac{5}{8}.$$

Therefore, for $y \geq 5k$, $\Pr(Y \geq y)$ can be upper bounded by a geometric series and we get

$$\Pr(Y \geq 5k) \leq \frac{8}{3}\Pr(Y = 5k) \;\leq\; \frac{8}{3}\binom{5k}{k}\frac{1}{2^{5k}} \leq \frac{8}{3}\left(\frac{5ek}{2^5 k}\right)^k \underset{(k\geq 2)}{\leq} \frac{1}{2^k}.$$

Adding the $2k - 1$ vertices where processes are split completes the proof. ◻

Lemma 12. *The step complexity of the first invocation of* STORE *requires* $\mathrm{O}(k)$ *steps in expectation and with high probability.*

Proof. A process visits at most $c \log n$ vertices in the randomized structure. With probability $1 - k/n^c$ at most $4\sqrt{k}$ processes enter the first backup structure and hence stop there. Applying Lemma 6, we conclude that with high probability the step complexity of the first store is linear in k. For the expectation we get

$$\begin{aligned}\mathrm{E}[\text{STORE}] &\le \left(1-\frac{k}{n^c}\right)(c\log n+2k)+\frac{k}{n^c}(c\log n+2\log 4\sqrt{n}\cdot\frac{4\sqrt{n}}{\log 4\sqrt{n}}+1)\\ &= \mathrm{O}(k).\end{aligned}$$

□

Lemma 13. *An invocation of* COLLECT *requires* $\mathrm{O}(k)$ *steps with high probability and in expectation. In any case, the step complexity of* COLLECT *is* $\mathrm{O}(k^2+k\log n)$.

Proof. Let $\mathcal{A}$ be the event that more than $7k$ nodes are marked in the random splitters tree. By Lemma 11, $\Pr(\mathcal{A}) \le 1/2^k$. Further, let $\mathcal{B}$ be the event that more than $4\sqrt{k}$ processes enter the cascaded trees backup structure. By Lemma 10 we have $\Pr(\mathcal{B}) \le k/n^c$ and therefore $\Pr(\mathcal{A}\cup\mathcal{B}) \le 1/2^k + k/n^c$. Hence, with probability at least $1 - 1/2^k - k/n^c$, the step complexity of a collect is at most $7k + (4\sqrt{k})^2/\log n = \mathrm{O}(k)$.

We compute the expected step complexity of a COLLECT operation in each of the three data structures separately. By linearity of expectation, we can sum up those results and get the total expected step complexity. Let S_T, S_C, and S_A denote the number of steps of a COLLECT operation, performed on the randomized splitter tree, the cascaded trees, and the array, respectively. By Lemma 11, we immediately have $\mathrm{E}[S_T] \le 3k$. For the cascaded trees structure we get

$$\begin{aligned}\mathrm{E}[S_C] \le{}& \left(1-\frac{k}{n^c}\right)\cdot\frac{(4\sqrt{k})^2}{\log n}+\sum_{j=4\sqrt{k}}^{\min(k,4\sqrt{n})}\mathrm{O}\left(\frac{j^2}{\log n}\cdot\frac{6k^2}{\left(j-\sqrt{k}\right)^2 n^{3/2}}\right)\\ &+\mathrm{O}\left(\frac{6k^2}{n(4\sqrt{n}-\sqrt{n})^2\cdot n^{3/2}}\cdot\left(\sqrt{n}\right)^3\right)\\ \le{}& \mathrm{O}\left(\frac{k}{\log n}+\frac{\min(k,\sqrt{n})}{\sqrt{n}}\cdot\frac{k^2}{n}+k\right)=\mathrm{O}(k),\end{aligned}$$

where we applied Lemma 8, Lemma 10 and the fact that the number of nodes in the cascaded trees structure is $\mathrm{O}((\sqrt{n})^3)$. For the second backup structure, the linear array, we get $\mathrm{E}[S_A] \le k/n^c \cdot n = \mathrm{O}(k/\sqrt{n})$. Summing up, we get $\mathrm{E}[\text{COLLECT}] = \mathrm{O}(k)$.

The worst-case number of vertices marked in the binary tree of randomized splitters is $\mathrm{O}(k\log n)$ because each process can mark at most $3/2\log n$ vertices. The step complexity in the cascaded-tree structure is at most $k^2/\log n$ by Lemma 7. If the linear array is accessed, the step complexity is $\mathrm{O}(n)$. However, this can only happen if $k > 4\sqrt{n}$, and therefore the lemma follows. □

5 Randomized Construction for Deterministic Collect

In this section, we show that instead of having processes, which have access to a random source, it is also possible to have a pre-computed ramdom splitter structure and keep the processes themselves deterministic. The random structure upon which the STORE and COLLECT is performed is constructed in a pre-processing phase. It is a random directed graph with n^3 vertices and out-degree 2 at all vertices. To each of the vertices, there is an deterministic splitter (cf. Section 3) associated. That is, we are given n^3 vertices, each of which chooses two random successors among the other vertices. The two successors of a vertex v are associated with **left** and **right** of v's splitter. One of the vertices is singled out and called the *root*.

The processes traverse the data structure as described in Section 3.1. Additionally, each process counts the number of visited splitters. If this number exceeds n, the process immediately leaves the random data structure and enters the backup structure. In the backup structure the process will then raise a flag at the root, indicating that the backup structure has been used, and then traverse the cascaded tree structure, as described in the Section 3.2.

To perform a COLLECT a process traverses the part of the random data structure containing marked vertices by simply visiting the children of a marked vertex in DFS order. The process furthermore memorizes which vertices it has already accessed and will not access a vertex twice. Additionally, it checks whether the flag in the backup structure is raised and, if that is the case, collects the values in the backup structure as described in the previous section.

To prove the correctness and complexity of the algorithm, we will proceed as in the previous section. Let k be the number of processes that call STORE at least once.

We have n^3 vertices in the randomized structure and $n^{\gamma+1}/\log n$, $\gamma > 1$ vertices in the backup structure. With $\gamma \leq 2$, we get the following lemma.

Lemma 14. *The memory complexity is* $\mathrm{O}(n^3)$.

Lemma 15. *The probability that a process enters the backup structure is at most* $\mathrm{O}(1/n)$.

Proof. A process traverses the data structure according to the values it is given in the splitters and leaves the random structure if it accessed more than n vertices. We want to show that, with high probability, the marked part of the data structure is a tree (that is, we do not have a cycle) and consequently a process stops with high probability after at most $k \leq n$ steps (see Lemma 2). Taking into account that by Lemma 3 in a tree at most $2k-1$ vertices are being marked, this leaves us to prove that the first $2k-1$ visited vertices are distinct and hence do not form a cycle with high probability.

We may model the way how the data structure is traversed by the processes as a 'balls-into-bins' game, since the children of a vertex were chosen uniformly at random. The number of balls is $2k-1$ and the number of bins is $N = n^3$. If we let C be the number of balls ending up in a bin containing more than one ball, by Lemma 8 the probability that C is at least one can be estimated as

$$\Pr(C \geq 1) \leq \frac{6(2k-1)^2}{(1-k^2/n^3)^2 n^3} \leq \mathrm{O}(1/n).$$

□

Lemma 16. *The step complexity of the first invocation of* STORE *requires expected and with high probability* $\mathrm{O}(k)$ *steps.*

Proof. By the previous lemma we know that with high probability the marked subgraph of our randomized data structure is a tree and hence, by Lemma 2, a store takes at most $k-1$ steps. Since a process makes at most n steps in the randomized structure and, by Lemma 6, k steps in the cascaded tree structure, we have:

$$\mathrm{E}[\text{STORE}] \leq \left(1-\frac{1}{n}\right)\cdot k + \frac{1}{n}\cdot(n+k) = \mathrm{O}(k).$$

□

Lemma 17. *With high probability and in expectation the step complexity of* COLLECT *is* $\mathrm{O}(k)$. *In any case, the step complexity is at most* $\mathrm{O}(n^2)$.

Proof. The collect time in the backup structure is at most $n^2/\log n$ and the processes leave the randomized structure after at most n steps. Hence, the step complexity of the COLLECT will never exceed $\mathrm{O}(n^2)$.

With probability $\left(1-\frac{1}{n}\right)$ the marked data structure is a tree and no process enters the backup structure. Hence, we can apply Lemma 3 and the step complexity of a collect is with high probability at most $(2k-1)$. If it enters the backup structure, a collect costs by Lemma 7 at most $k^2/\log n$ and furthermore at most kn vertices are marked in the randomized structure. Hence, for the expected step complexity we get

$$\mathrm{E}[\text{COLLECT}] \leq \left(1-\frac{1}{n}\right)\cdot(2k-1) + \left(\frac{1}{n}\right)\cdot\left(kn+\frac{k^2}{\log n}\right) \;=\; \mathrm{O}(k).$$

□

6 Conclusions

We presented new deterministic and randomized adaptive collect algorithms. Table 1 compares the three 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 contention definitions which are more fine-grained, such as point contention. The *point contention*

Algorithm	Step Complexity COLLECT	Step Complexity STORE	Memory Complexity	
triangular matrix [14]	$\mathrm{O}(k^2)$	$\mathrm{O}(k)$	$\mathrm{O}(n^2)$	deterministic
tree [8]	$\mathrm{O}(k)$	$\mathrm{O}(k)$	$\mathrm{O}(2^n)$	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)	$\mathrm{O}(k)$	$\mathrm{O}(k)$	$\mathrm{O}(n^{3/2})$	randomized
randomized graph (Sec. 5)	$\mathrm{O}(k)$	$\mathrm{O}(k)$	$\mathrm{O}(n^3)$	randomized

Table 1. Summary of the complexities achieved by different collect algorithms. Note that the kind of randomization used in the two randomized algorithms is inherently different. For the algorithm of Section 4, the processes need access to a random source while the algorithm of Section 5 works on a random graph which can be precomputed.

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, 4, 7].

Our paper shows that it is possible to perform a COLLECT operation in $O(k)$ time with polynomial memory using randomization. We believe that there is no deterministic algorithm, using splitters, that achieves linear COLLECT and polynomial memory. 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 (PODC)*, 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 (FOCS)*, pages 262–272. IEEE Computer Society Press, 1999.

[3] Y. Afek, G. Strupp, and D. Touitou. Long-lived and adaptive splitters and applications. volume 15, pages 444–468, 2002.

[4] 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 (PODC)*, pages 71–80, 2000.

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

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

[7] H. Attiya and A. Fouren. Algorithms Adaptive to Point Contention. *Journal of the ACM (JACM)*, 50(4):444–468, July 2003.

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

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

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

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

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

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

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