Incremental Calculation for Fully Adaptive Algorithms

Hagit Attiya and Idan Zach
Department of Computer Science
The Technion, Haifa 32000, Israel
hagit@cs.technion.ac.il, idanz@cs.technion.ac.il

***Abstract*— The step complexity of *fully adaptive* (FA) algorithms depends only on the contention during an operation, when counting both *local computation* and accesses to shared registers. This paper contributes to the design of efficient fully adaptive algorithms by specifying and implementing two generic objects, *Gather&*f and *Collect&*f.**

A Gather&f object returns the value of applying a function f on the information previously stored in the object. In order to reduce the local step complexity, our implementation of a Gather&f calculates f *incrementally*, as values are stored in the object. This implementation uses only read and write operations on the shared memory. It has $O(k)$ *local* and shared step complexity for calculating f and $O(k^2)$ for storing information, where k is the point contention during the operation's execution interval.

A Collect&f further guarantees that the value of f returned by a later collect&f operation does not reflect fewer store operations than a (strictly) earlier collect&f operation. A simple implementation has $O(k^2)$ local and shared step complexity for calculating f and for storing information. For common applications where collect&f is repeatedly invoked, the paper presents an efficient object called *ECollect&*f. In this object, storing information takes a single step and calculating f has $O(k)$ local and shared step complexity.

To demonstrate the applicability of Gather&f and ECollect&f, we use them to improve the local and shared step complexity of atomic snapshot and immediate snapshot to $O(k^2)$ and $O(k^3)$, respectively.

I. Introduction

Distributed algorithms are designed to accommodate a large number of potential participants; yet, the step complexity of an operation should depend only on the number of participating processes. Ideally, the step complexity should be a function only of the *point contention*—the maximum number of processes executing concurrently at some point—during the operation.

There are algorithms whose *shared* step complexity—the number of accesses to shared registers—is adaptive to point contention [2], [3], [4], [8]. However, except for [4], the *local* step complexity of these algorithms is not a function of the point contention. A typical situation happens, for example, when operations store their values in some data structure, requiring a later computation of the maximum value to access the whole data structure. In this situation, the local step complexity of an operation is proportional to the *total contention*—the number of processes that ever participated in the algorithm—even if its point contention is one, i.e., the operation executes in isolation.

This paper is devoted to the design of efficient *fully adaptive* (FA) algorithms, whose local and shared step complexity depends only on the point contention. We specify and implement two generic objects, *Gather&*f and *Collect&*f, as modular mechanisms for designing fully adaptive algorithms; the Collect&f object is then optimized for repeated invocations. These objects are employed to significantly reduce the local and shared step complexity of atomic and immediate snapshots.

A Gather&f object allows processes to *store* information and to *retrieve* the result of applying the function f to the information stored in the object, before or possibly during the calculation of f. (A precise definition appears in Section II.) For example, the Gather&max object returns the largest value stored, and is equivalent to the *pile* object [4]. Another example is the Gather&$cluster$ object, which maintains the set of currently active processes; it implements an *active set* object [4].

The key for reducing local computation costs is to *reuse* previous calculations, by storing their result instead of the raw data they are based on. We present (in Section III) an implementation of a Gather&f object, with $O(k)$ local and shared step complexity for calculating f and $O(k^2)$ local and shared step complexity for storing information, where k is the point contention during the operation's execution interval. The implementation extends the idea of incremental calculation from the pile object of Afek et al. [4] and fits it into the collect algorithm of Attiya and Fouren [8]. As a special case, our implementation improves the local and shared step

complexity of the pile object from $O(k^3)$ to $O(k^2)$.

A simple extension of the Gather&f implementation yields a Collect&f object, where the value of f returned by a collect&f operation is more up-to-date than (strictly) earlier collect&f operations. Unfortunately, the (local and shared) step complexity of calculating f is $O(k^2)$, which can be quite prohibitive when f is repeatedly calculated. For such applications, we present (Section IV) an efficient ECollect&f object in which storing information takes only a single step, while collect&f has $O(k)$ local and shared step complexity. The implementation uses the Gather&max and active set objects.

To demonstrate the applicability of the Gather&f and ECollect&f objects, we use them to improve the (local and shared) step complexity of atomic snapshots and immediate snapshots. *Atomic snapshots* [1], [5], [6] return "instantaneous" views of the shared memory and are an important tool in the design of shared-memory algorithms. *Immediate snapshots* [11] guarantee, in addition, that returned views are not much later than the previous update by the same process; they play a key role in the study of asynchronous computability [10], [13].

In Section V, we improve the (local and shared) step complexity of the atomic snapshot algorithm [4] to $O(k^2)$ and of the immediate snapshot algorithm [4] to $O(k^3)$. (The relationships between algorithms presented in this paper are summarized in Figure 1.)

Table I compares the step complexity bounds of our atomic and immediate snapshots algorithms with previous adaptive algorithms. Attiya, Fouren and Gafni [9] present atomic and immediate snapshot algorithms; however, their step complexity adapts only to the total contention. Afek et al. [4] present fully adaptive atomic and immediate snapshot algorithms with $O(k^4)$ and $O(k^5)$ *local* and shared step complexity, respectively. Attiya and Fouren [8] implement atomic snapshots with $O(k^3)$ step complexity; however, the step complexity accounts only for shared memory accesses and their local step complexity is not adaptive to point contention.

II. The Gather&f and Collect&f Objects

A. Preliminaries

We assume a standard asynchronous shared-memory model of computation, following, e.g., [12]. A system consists of n processes, $p_1, \ldots, p_n$. A process can read from and write to registers that are either local to itself or shared by all processes.

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

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

An invocation of a high-level operation by a process causes the execution of the appropriate algorithm. The *execution interval* of an operation op_i by process p_i is the subsequence of the execution between the first event of p_i in op_i and the last event of p_i in op_i. Assume that the execution interval of an operation op_i by process p_i precedes the execution interval of an operation op_j of process p_j; namely, the last event of p_i in op_i appears before the first event of p_j in op_j. In this case op_i *precedes* op_j and op_j *follows* op_i, denoted $op_i \rightarrow op_j$.

When objects are implemented in a hierarchical manner, executions may contain several *levels* of operations. A *high-level* operation hop consists of a sequence of *low-level* operations: $lop_1, \ldots, lop_n$, where the first one, lop_1, starts when hop starts and the last one, lop_n, completes when hop completes.

Process p_i is *active* at the end of α', a finite prefix of an execution α, if α' includes an invocation of a high-level operation op by p_i without the matching return. The *point contention* at the end of α', denoted $PntCont(\alpha')$, is the number of active processes at the end of α'.

Consider a finite interval β of an execution α; we can write $\alpha = \alpha_1\beta\alpha_2$. The point contention during β, denoted $PntCont(\beta)$, is the maximum contention in prefixes $\alpha_1\beta'$ of $\alpha_1\beta$, namely

$$\max\{|Cont(\alpha_1\beta')| : \alpha_1\beta' \text{ is a prefix of } \alpha_1\beta \} .$$

B. Gather&f and Collect&f: Definitions

In this section, we define the Gather&f and Collect&f objects. Consider an *associative*, *commutative* and *idempotent*[1] function f. In both objects, a process can store its value in a shared memory, and retrieve the result of applying function f to the values stored in the shared memory. Instances of the object are differentiated by the function f that they support, e.g., *max* or *union*.

1) The Gather&f Object: The object handles *tagged values*, which are $\langle tag, value \rangle$ pairs and provides two operations: put&f stores its parameter in the object, while gather&f returns the result of applying f to all the tagged values stored in the object before or possibly during the operation.

[1]A function is idempotent if applying it multiple times on same value is equal to a single application on the same value.

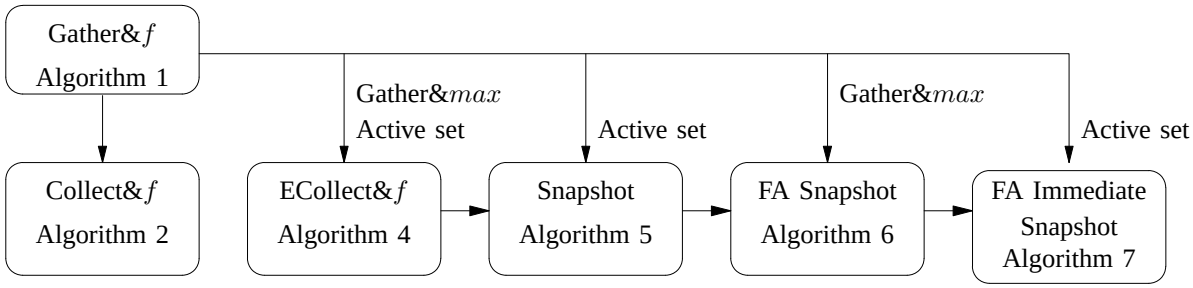


Fig. 1. Relationships between algorithms presented in this paper.

Problem	*Reference*	*Operations Counted*	*Step Complexity*
atomic snapshot	[8]	shared only	$O(k^3)$
	[4]	shared+local	$O(k^4)$
	(this paper)	shared+local	$O(k^2)$
immediate snapshot	[9]	shared only	$O(K^3)$ (K = total contention)
	[4]	shared+local	$O(k^5)$
	(this paper)	shared+local	$O(k^3)$

TABLE I

COMPARISON OF ATOMIC AND IMMEDIATE SNAPSHOTS ALGORITHMS.

A set W of put&f operations can be identified with their parameters; $f(W)$ denotes the result of applying f on the parameters of the operations in W. This is well-defined since f is associative and commutative.

A gather&f operation *op* returns a tagged value $\langle tag, res\rangle$ such that the following condition holds: There exists a set W of put&f operations that includes all operations preceding *op* and does not include any operation following *op*, such that $\langle tag, res\rangle = f(W)$. For a put&$f$ operation $op' \in W$, we denote $op' \triangleright f(W)$.

2) The Collect&f Object: Like the Gather&f object, this object provides two operations: put&f stores its parameter in the object, while collect&f returns the result of applying f to all the values stored in the object before or possibly during the operation. In this object, later collect&f operations are at least as updated as previous ones.

Formally, consider two collect&f operations op_1 and op_2 such that $op_1 \rightarrow op_2$; if op_1 and op_2 return $f(W_1)$ and $f(W_2)$, respectively, then $W_1 \subseteq W_2$.

C. Examples

We present several examples of important objects that reduce to Gather&f. Thus, a fully adaptive algorithm for a generic Gather&f object allows to efficiently implement several other objects, provided we have a locally efficient algorithm for computing f.

1) Example 1: Consider the problem of atomically computing an aggregate function, such as max or min, of a set of values, while these values are dynamically updated by processes. Each of these problems trivially reduces to a Gather&f object, where f is the appropriate aggregation function.

2) Example 2: The simple *collect* object supports two operations: a store(v) operation of p_i declares v as the latest value for p_i, and a collect operation returns a view[2] containing the latest values stored by active processes. A collect operation *cop* returns a view V satisfying the following condition, for every process p_j:

Validity: If $V(p_j) = \perp$, then no store operation of p_j precedes *cop*; if $V(p_j) = v \neq \perp$ then v is the value of a store operation *sop* of p_j that does not follow *cop*, and there is no other store operation *sop'* of p_j that follows *sop* and precedes *cop*.

This means that *cop* should not read from the future or miss a preceding store operation. Moreover, if *cop* follows another collect operation *cop'*, then *cop* returns a view that is more up-to-date. To capture this notion formally, we define a partial order on views: $V_1 \preceq V_2$ if for every process p_i such that $\langle p_i, v_i^1\rangle \in V_1$, we have $\langle p_i, v_i^2\rangle \in V_2$, and $v_i^2 \geq v_i^1$.

Regularity: Assume a collect operation *cop* by p_i returns V_1, and a collect operation *cop'* by p_j returns V_2. If *cop* precedes *cop'*, then $V_1 \preceq V_2$.

It is simple to see that the collect problem reduces to

[2]A *view* is a set of process-value pairs, $V = \{\langle p_i, v_i\rangle, \ldots\}$, without repetitions of processes. $V(p_j)$ is v_j if $\langle p_j, v_j\rangle \in V$ and $\perp$, otherwise.

a Collect&$union$ object.

3) Example 3: An *active set* object [4] allows to obtain the set of currently active processes, by providing three operations: joinSet signs a process into the active set; leaveSet signs a process out of the active set; getSet returns the set of processes signed in the active set. The set P returned by getSet operation gop_j of process p_j must satisfy the following conditions:

1) If joinSet operation by process p_i precedes gop_j and no leaveSet operation by p_i starts before gop_j ends, $p_i \in P$.
2) If leaveSet operation by process p_i precedes gop_j and there is no following joinSet operations by p_i, or there are no joinSet operations by p_i, then $p_i \notin P$.

The active set object can be reduced to a Gather&$cluster$ object, where *cluster* is an extension of the *union* function, so that a process changing its value in the view to $\perp$ is removed from the view. Thus, joinSet is mapped to put&$cluster(\langle 0, p_i \rangle)$, leaveSet is mapped to put&$cluster(\langle 0, \perp \rangle$ and getSet is mapped to gather&$cluster$.

III. Implementations for Gather&f and Collect&f objects

A. Informal Description

A simple implementation of a Gather&f object is to use an ordinary Gather object [8], satisfying the *validity* property (defined for the collect object in Section II-C.2). A gather&f operation simply invokes a gather operation and applies f on the view it returns. The main difficulty is to calculate the new result with adaptive local step complexity. If the result is calculated from scratch each time, all stored values should be read. Thus, the local step complexity is not bounded by the point contention, but by the total contention (at best).

In order to bound the number of local steps, we incrementally calculate the result and save it for later use, as in the pile algorithm [4]. We take a gather algorithm [8] and change its behavior so that instead of storing the data itself, it calculates the partial result incrementally and saves it for later use.

In the gather algorithm we use [8], a process tries to join one of $2k-1$ groups ordered in a column; each group maintains a view containing all the values of its members. A group is dynamically managed with a *sieve*, a data structure that allows processes to exchange information; a sieve captures the information of at least one of the processes accessing it concurrently (see the appendix).

In our Gather&f implementation, the processes in the group apply f to the view containing the values of its members and to partially computed results. Interim results are saved in a shared memory location associated with the group. Later operations do not recalculate the result, but only combine the previously-calculated value with new information that was added since the last calculation.

To restrict the effects of high point contention during a put&f operation, processes propagate (*bubble up*) new results to the top of the array. Otherwise, put&f encountering high contention may store a new value in an entry with large index s_0, and a subsequent gather&f will have to access many entries, even if its point contention is low. Bubble-up, presented in [3] and used also in [4], [8], allows gather&f to find the new values close to the top of the array, when contention is low.

B. The Algorithm

The algorithm uses the following data structures:

- A column of sieves, numbered $1..2n-1$;
- An array *Res*$[1..2n-1]$, $2n-1$ atomic registers of type $\langle tag, result \rangle$; The s^{th} entry of this array contains the result of applying f on the values of all past and current candidates in sieve s.
- An array *last*$[1..2n-1]$, $2n-1$ atomic registers of type pid, each initialized to $n+1$;
- An array $C[1..n+1][1..2n-1]$, $2n^2$ atomic registers of type $\langle tag, result \rangle$, each initialized to $\langle 0, \emptyset \rangle$;

The function f also calculates a *tag* value, which may used by the application as meta-data. In Gather&max, for example, the *tag* is a sequence number which is used for comparison between values. The initial values of both *tag* and *result* should be values that do not affect on the calculation. That is, for any set of values (including the empty set), W, *initial values* $\triangleright f(W)$.

Algorithm 1 presents the pseudo-code of the Gather&f object; lines changed from [8] are numbered in **bold**. The interface to the sieve is with procedures open, enter and exit [8]. Informally, open tells whether the sieve is busy, that is, other processes are inside the current copy of this sieve, enter tries to store information in a non-busy sieve, while exit releases the sieve. The sieve's properties guarantee that if some processes access copy c of sieve s, then at least one of them is a winner (see the appendix).

A put&f operation stores *value* as part of the result of all candidates in sieve s in an entry s of array *Res* and bubbles the result of previous values up to the top of the array. The first part of put&f has similar outline as previous applications of the sieve [7], [8]. A process

Algorithm 1 Gather&f: code for process p_i.

```
Shared:
    Res[1..2n − 1] : array of pairs, initially ⟨0, ⊥⟩
    last[1..2n − 1] : array of process id's, initially n + 1
    C[1..n + 1][1..2n − 1] : array of pairs, initially ⟨0, ∅⟩

void procedure put&f(⟨tag, value⟩)
1:  s = 0                          // try to win sieve 1, 2, ..., 2k − 1
2:  repeat
3:      s++;
4:      if open(s) then
5:          if p_i ∈ W = enter(s, ⟨tag, value⟩) then
                                                   // p_i wins sieve s
6:              for every ⟨id_j, ⟨tag_j, v_j⟩⟩ ∈ W
7:                  Res[s] = f(Res[s], ⟨tag_j, v_j⟩)
    // applying f on the previous result and the new value
8:          exit(s)
9:  until p_i ∈ W

10: while (s ≥ 1)                  // bubble up the new value
11:     C[id_i][s] = ⟨0, ⊥⟩
                    // announce you are accessing entry s
12:     last[s] = id_i
13:     C[id_i][s] = choose&f(s)
                    // get the calculated value below sieve s
14:     s − −;

⟨tag, result⟩ procedure choose&f(s: int)
15: q = last[s]
16: ⟨tag, tmp⟩ = C[q][s]
17: if (tmp ≠ ⊥) then return (⟨tag, tmp⟩)
        // another process accesses entry s concurrently
18: ⟨tag', v'⟩ = Res[s]
19: ⟨tag, v⟩ = choose&f(s + 1)          // recursive call
20: return (f(⟨tag, v⟩, ⟨tag', v'⟩))

⟨tag, result⟩ procedure gather&f()
21: return (choose&f(1))
```

p_i goes through the column of sieves, starting from the top, until it wins one of them. If p_i wins sieve s, it writes the result of the current value of *Res*[s] and the values currently stored in sieve s into *Res*[s]. The sieve properties guarantee that *Res*[s] is the result of all values previously stored in sieve s.

To bubble up, a process goes from entry s up to entry 1 and for each entry $s' = s, \ldots, 1$, it recursively calculates function f on the values stored in the part of the array below entry s, and stores the result in its private register associated with entry s.

In choose&$f(s)$, process p_i reads q from *last*[s] (Line 15) and then reads $\langle tag, tmp\rangle$ from $C[q][s]$. If $tmp \neq \perp$, then p_i returns *tmp*; otherwise, p_i calls choose&$f(s+1)$, and calculates the result of the return value and the value in *Res*[s].

C. Proof of Correctness

We adjust the correctness proof for the collect object [8] to consider the aggregate value of f on the stored values, instead of the values themselves. This shows that the incremental calculation in our algorithm correctly applies f to the information stored in the object.

We say that a put&f operation *crosses* sieve x' when it writes to *last*[x'] (Line 12). The proof of the next lemma follows the arguments of [8, Lemma 6.2].

Lemma 3.1: If a put&f operation pop_r of process p_r with input $\langle tag_r, value_r\rangle$ crosses entry $s \geq 1$ before a choose&$f(s)$ operation gop_g of process p_g returning pair $\langle tag_g, result_g\rangle$ starts, then $value_r \triangleright result_g$.

As in [8, Lemma 6.3], we can use Lemma 3.1 to prove that Algorithm 1 satisfies the properties of the Gather&f object.

The next lemma bounds the number of entries p_i accesses during gather&f and it follows [8, Lemma 6.5].

Lemma 3.2: The maximal entry accessed by process p_i performing a choose&f operation gop_i is $3k$, where k is the point contention during gop_i's execution interval.

By Lemma 3.2, choose&f operation accesses at most $3k$ entries, and its step complexity is $O(k)$ (see the appendix).

In Lines 1-9 of put&f, a process accesses $s_0 \leq 2k-1$ sieves, implying that the step complexity of this stage is $O(k^2)$. In the bubbling up stage of put&f, a process performs choose&f(s) for $s = s_0 - 1, \ldots, 1$; thus, the step complexity of this stage is also $O(k^2)$.

Since gather&f calls choose&f, its step complexity is $O(k)$.

Theorem 3.3: Algorithm 1 implements a Gather&f object with $O(k^2)$ local and shared step complexity for put&f and $O(k)$ local and shared step complexity for gather&f, where k is the point contention during the operation's execution interval.

D. Algorithm for Collect&f Object

As in [8], the collect&f procedure first calls gather&f and then calls put&f to store the result it has obtained, in order to guarantee the collect property. Algorithm 2 presents the additional pseudo-code for the Collect&f object.

The proof of the next lemma follows [8, Lemma 6.4].

Algorithm 2 Collect&f: additional code for process p_i.

$\langle tag, result\rangle$ procedure collect&f()
1: $\langle tag, result\rangle$ = gather&f()
2: put&f($\langle tag, result\rangle$)
3: return ($\langle tag, result\rangle$)

Lemma 3.4: Algorithm 2 satisfies the properties of the Collect&f object.

Since collect&f calls put&f, its local and shared step complexity is $O(k^2)$.

Theorem 3.5: Algorithm 2 implements a Collect&f object with $O(k^2)$ local and shared step complexity for both put&f and collect&f operations, where k is the point contention during the operation's execution interval.

E. Applications

This section presents applications of Gather&f object, which differ in the function f that they calculate.

1) Gather&max: A Gather&max object returns the maximum value previously stored in the object, that is, the pair $\langle tag, value\rangle$ with the maximum *tag*. In this case, f is a max function that compares two scalars (two *tag* values), and it can be implemented in a single local step. Thus, the implementation of this object requires $O(k)$ steps for gather&f and $O(k^2)$ steps for put&f, where k is the point contention and local steps are also counted.

2) Gather&$union$: A Gather&$union$ object returns a $\langle tag, view\rangle$, where *view* contains 3-tuples of the form $\langle pid, value, seq\rangle$. where *pid* is a process's identifier, *value* is the value itself and *seq* is a sequence number (every value is tagged with a sequentially increasing label, called sequence number), which is used to pick the latest value for each process. The *tag* contains the sum of all the sequence numbers in *view*, and it is used later to make the efficient comparisons between views.

The implementation of the *union* function requires some care since a view might include many values, but we still want to achieve low local step complexity. To do so, Algorithm 3 merges the values of the active processes with the last view stored in the object, by scanning only the new values (the values of the active processes). The local computation in the *union* function is dominated by a loop over the new values.

Collect&$union$, namely, an ordinary collect object, can be easily implemented by a Collect&f object, in the same manner that Gather&$union$ is implemented by Gather&f.

Algorithm 3 Gather&$union$: additional code for process p_i.

$\langle tag, view\rangle$ procedure union($\langle tag_1, newV\rangle$,$\langle tag_2, srcV\rangle$)
// For every process, take latest entry from *srcV* , *newV*.
// Size of *newV* is bounded by the contention.
1: for every $\langle pid_1, v_1, seq_1\rangle \in$ *newV*
2: if $\exists\langle pid_1, v_2, seq_2\rangle \in$ *srcV* then
3: if $seq_1 > seq_2$ then
4: $srcV = srcV \cup \langle pid_1, v_1, seq_1\rangle \setminus \langle pid_1, v_2, seq_2\rangle$
5: $tag_2 = tag_2 + seq_1 - seq_2$
else // no tuple for pid_1 in *srcV*
6: $srcView = srcV \cup \langle pid_1, v_1, seq_1\rangle$
7: $tag_2 = tag_2 + seq_1$
8: return ($\langle tag_2, srcV\rangle$)

3) Active set: An active set object, which returns a set of active processes can be supported by a Gather&$cluster$ object, defined in Section II-C.3. Care is required to maintain the set of active processes for computing *cluster*; in order to reduce the local step complexity we need to explicitly remove the identifiers of processes that are not active (for a leaveSet operation), without missing any information. The implementation of the *cluster* function is similar to the implementation of the *union* function, but we manage a clustered set of the identifiers, for processes with non-$\perp$ value in the full view. Once the original view is updated the process identifier is added or removed from the clustered set, accordingly.

IV. The ECollect&f object

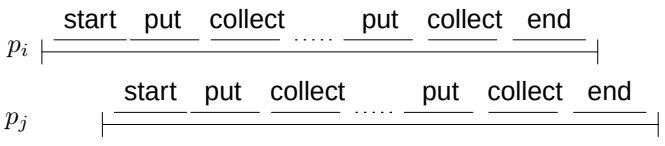
Several algorithms repeatedly invoke put and/or a collect operations within a single high-level operation. The repetitions of expensive put and collect operations are the major factor in the step complexity of those algorithms.

In this section, we introduce a new efficient Collect&f object, ECollect&f, which is optimized for performing many put&f and collect&f operations within a single high-level operation. A put&f operation requires only a single step and a collect&f operation has $O(k)$ local and shared step complexity, where k is the point contention during the high-level operation's execution interval.

We reduce the cost of storing and collecting information by requiring costly coordination only when a process starts or completes a high-level operation. After announcing their participation at the beginning of the operation, active processes update their information in a simple shared register; before completing their operation, processes record their latest information at a readily

available shared location. To collect current information, a process obtains the set of active processes and reads from their registers, then applying function f on the values of the active processes and the latest result previously calculated and stored.

The ECollect&f object is similar to the Collect&f object, except it provides *four* operations: start, end, put&f and collect&f. The put&f and collect&f operations are the same as in the Collect&f object, but it is assumed that start is called before put&f and collect&f are called, and that end is called before the process completes the high-level operation.



The start operation indicates that the process starts a high-level operation, which should be coordinated with other concurrent processes. The end operation indicates that the process has just completed the high-level operation, and its current result should be saved.

A. Implementation

The implementation of ECollect&f object is based on two instances of the Gather&f object: Gather&max and active set. The Gather&max is maintains views of high-level operations that have completed, while the active set object captures concurrent processes.

The start operation signs a process into the active set. In a put&f operation a process only writes its current values to an atomic register, $R[p_i]$; write operations are tagged with a sequence number, in order to observe between new and old values. A collect&f operation obtains the active set of processes and reads from their registers, $R[p_j]$. Then it applies the function f to the values of the active processes and the values of non-active processes it gets from the Gather&max object. The end operation writes its information to the Gather&max object, and then signs out of the active set.

Algorithm 4 presents the pseudo-code of the ECollect&f object.

Algorithm 4 ECollect&f: code for process p_i.

```
Shared:
    Gm : Gather&max object
    AS : active set object        // set of active processes
    R[1..n] : array of pairs, initially ⟨0, ⊥⟩

void procedure start()
1:  AS.joinSet()                                  // sign-in

void procedure end()
2:  Gm.put&max(collect&f())       // store current view
3:  AS.leaveSet()                                 // sign-out

void procedure put&f(value)
4:  R[p_i] = ⟨R[p_i].seq + 1, value⟩

⟨tag, result⟩ procedure collect&f()
5:  actV = AS.getSet()
6:  newV = {⟨p_j, R[p_j].value, R[p_j].seq⟩ |
                          p_j ∈ actV}
7:  ⟨tag, result⟩ = Gm.gather&max()
8:  return(f(newV, ⟨tag, result⟩))
```

B. Proof of correctness

We prove that ECollect&f provides the properties of a *Collect&f* object, assuming that start and end encapsulate the calls to collect&f and put&f.

It is obvious that the pair $\langle tag, result\rangle$ returned by a collect&f operation cop_i of process p_i does not contain values written by put&f operations follow cop_i. The next lemma implies that $\langle tag, result\rangle$ includes the latest value stored by each process p_l before cop_i.

Lemma 4.1: Assume a collect&f operation cop_i by p_i returns $\langle tag, result\rangle$. If a put&$f(\langle tag_l, value_l\rangle)$ operation pop_l by p_l precedes cop_i, then $\langle tag_l, value_l\rangle \triangleright \langle tag, result\rangle$.

Proof: Consider the final put&max (Line 2) operation of p_l. If it completes before cop_i starts gather&max (Line 7) operation, then cop_i includes $\langle tag_l, value_l\rangle$, by the properties of the Gather&f object. Otherwise, the corresponding put&max (Line 2) operation of p_l completes after the gather&max (Line 7) operation in cop_i starts. Note that p_l calls leaveSet (Line 3) after it completes put&max (Line 2); it follows that the getSet (Line 5) operation of cop_i completes before leaveSet (Line 3) by p_l starts, since the getSet operation precedes the gather&max operation. Thus, p_l is in the active set observed by cop_i, which reads $\langle tag_l, value_l\rangle$, or a later value, from $R[p_l]$. ■

Lemma 4.2: Assume a collect&f operation cop_i by p_i returns $\langle tag_1, result_1\rangle$ and a collect&f operation cop_j by p_j returns $\langle tag_2, result_2\rangle$. If cop_i precedes cop_j, then $result_2 \triangleright result_1$.

Proof: For an arbitrary pair $\langle tag_l, value_l\rangle \triangleright result_1$, we consider two cases:

Case 1: $\langle tag_l, value_l\rangle$ is read from the Gather&max

object. Consider the corresponding put&max (Line 2) operation of p_l. If it completes before *cop*$_j$ starts the gather&max (Line 7) operation, then *cop*$_j$ includes $\langle tag_l, value_l \rangle$, or a later value. Otherwise, the corresponding put&max (Line 2) operation of p_l completes after the gather&max (Line 7) operation in *cop*$_j$ starts. Note that p_l calls leaveSet (Line 3) after it completes put&max; it follows that the getSet (Line 5) operation of *cop*$_j$ completes before leaveSet (Line 3) by p_l starts, since the getSet operation precedes the gather&max operation. Thus, p_l is in the active set observed by *cop*$_j$, which reads $value_l$ from $R[p_l]$.

Case 2: $\langle tag_l, value_l \rangle$ is read from $R[p_l]$. This means that p_l writes to $R[p_l]$, after completing its recent joinSet (Line 1), before *cop*$_i$ completes. Therefore, p_l completes its joinSet (Line 1) before *cop*$_j$ starts, since *cop*$_j$ follows *cop*$_i$. If p_l is in the active set obtained by *cop*$_j$, then p_j reads $\langle tag_l, value_l \rangle$ or a later value from $R[p_l]$ and the claim follows. Otherwise, it must be that p_l starts leaveSet (Line 3) before the end of the getSet (Line 5) operation of *cop*$_j$. Then, p_l completes put&max (Line 2) before *cop*$_j$ starts gather&max (Line 7), which implies that $result_2$ includes $\langle tag_l, value_l \rangle$, or a later value. ■

Finally, we compute the step complexity of this algorithm. A start operation calls joinSet, therefore its local and shared step complexity is $O(k^2)$, where k is the point contention during the high-level operation's execution interval. A collect&f operation calls getSet, reads the registers of the active processes and then calls gather&max. Thus, its local and shared step complexity is $O(k)$. The step complexity of put&f operation is clearly constant. An end operation calls put&max and then leaveSet, therefore its local and shared step complexity is $O(k^2)$.

Theorem 4.3: The ECollect&f object has $O(k^2)$ local and shared step complexity for start and end operations, $O(k)$ for collect&f operation and $O(1)$ for put&f operation, where k is the point contention during the high-level operation's execution interval.

V. Improving the Step Complexity of Atomic Snapshots and Immediate Snapshots

A. Atomic Snapshots

The *atomic snapshot* problem[1], [5], [6] extends the collect problem by requiring views to look instantaneous; it supports two operations, update, which updates a new value, and scan which atomically obtains a view. In addition to the validity and regularity properties (defined in Section II-C.2), the returned views should satisfy the following condition:

Algorithm 5 Atomic snapshot [4]: code for process p_i.

```
Shared:
    EC : ECollect&union object
    B[1..n] : registers with ⟨view, view⟩, initially ⟨∅, ∅⟩
    AS : active set object        // set of active processes
Local:
    seq : integer, initially 0

⟨tag, view⟩ procedure scan()
1:  seq = seq + 1
2:  AS.joinSet()                                  // sign-in
3:  while (true) do
4:    ⟨tag_1, v_1⟩ = EC.collect&union()
5:    ⟨tag_2, v_2⟩ = EC.collect&union()
6:    if (tag_1 = tag_2 and v_1 = v_2) then
7:      AS.leaveSet()                             // sign-out
8:      return (⟨tag_2, v_2⟩)
9:    if ∃pid' s.t., ⟨pid', *⟩ ∈ v_1 ∪ v_2 and
                  ⟨pid, seq⟩ ∈ B[pid'].s then
10:     AS.leaveSet()                             // sign-out
11:     return (B[pid'].sc)

void procedure update(value)
0:  EC.start()              // start a high-level operation
1:  ⟨t, s⟩ = AS.getSet()
2:  ⟨t, sc⟩ = scan()
3:  B[pid] = ⟨s, sc⟩
4:  EC.put&union(value)
5:  EC.end()                // end the high-level operation
```

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

We can improve the *shared* step complexity of the snapshot algorithm [4] to $O(k^2)$, by using ECollect&$union$. In this algorithm, a process p_i repeatedly double collects until either two collects are identical or some process p_j was observed to change its value and its scan has started after p_i started. Pseudo-code, using ECollect&$union$, appears in algorithm 5.

The correctness of the algorithm follows as in [4], [8], since the only change is using the efficient collect object, ECollect&$union$, instead of an ordinary collect object.

A scan operation makes at most k iterations, each requiring $O(k)$ steps (the ECollect&$union$ has $O(k)$ step complexity), the step complexity of scan and therefore also of update is $O(k^2)$.

Theorem 5.1: Algorithm 5 implements the atomic snapshot object with $O(k^2)$ shared steps, where k is the point contention during the operation's execution interval.

Algorithm 6 Fully adaptive atomic snapshot [4]: code for process p_i.

```
Shared:
    S : atomic snapshot object        // from Algorithm 5
    Gm : Gather&max object

⟨tag, view⟩ procedure scan()
1:  ⟨tag_1, v_1⟩ = S.scan()
          // Size of v_1 is bounded by point contention
2:  ⟨tag_2, v_2⟩ = Gm.gather&max()
3:  return(union(⟨tag_1, v_1⟩, ⟨tag_2, v_2⟩))
                                  // union from Algorithm 3

void procedure update(value)
4:  S.update(value)  // register to the current snapshot
5:  ⟨tag, v⟩ = scan()
6:  Gm.put&max(⟨tag, v⟩)
7:  S.update(⊥)                                  // sign-out
```

Unfortunately, the local step complexity of Algorithm 5 is not adaptive to point contention, since the algorithm manipulates views containing values of currently non-participating processes, e.g., in a component-wise comparison (Line 6). By plugging our snapshot and Gather&max objects into the fully adaptive snapshot algorithm of [4], we achieve a fully adaptive snapshot algorithm with $O(k^2)$ local and shared step complexity. Algorithm 6 presents the pseudo-code of the fully adaptive snapshot algorithm.

In the scan operation, a process first accesses the snapshot object, which takes $O(k^2)$ steps, and then accesses the Gather&max, which takes $O(k)$ steps. Finally, it merges the two views in the same manner as in Algorithm 3. Therefore, the local and shared step complexity of scan operation is $O(k^2)$. The update operation calls scan, and accesses the Gather&max and snapshot objects, therefore its step complexity is also $O(k^2)$.

The correctness of the algorithm follows as in [4].

Theorem 5.2: Algorithm 6 implements the atomic snapshot object with $O(k^2)$ local and shared steps, where k is the point contention during the operation's execution interval.

B. Immediate Snapshots

The *Immediate snapshot* problem [11] provides a combined im-upscan operation, updating a new value and returning a view. In addition to the validity, regularity and comparability properties of the atomic snapshot problem, returned views should satisfy the next condition:

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

The local and shared step complexity of the immediate snapshot algorithm of [4] is improved to $O(k^3)$, by employing our fully adaptive algorithms for collect and atomic snapshot. The pseudo-code appears in Algorithm 7.

The correctness of the algorithm follows as in [4].

As was shown in previous immediate snapshot algorithms [4], [9], a process can descend at most k floors if it starts from the floor corresponding to *smallest* view containing its previous value.

The local and shared step complexity of accessing the active set object is $O(k^2)$, where k is the point contention. Since each process signs out of the active set object (Line 23), $|T| \leq k$ and there at most k iterations of the loop in Lines 5-7. The snapshot operations during *op* take $O(k^2)$ local and shared steps.

Since a process descends through at most $O(k)$ floor, in each one calling 1s-immss once with $O(k^2)$ step complexity, we get the next result.

Theorem 5.3: Algorithm 7 implements the immediate snapshot object with $O(k^3)$ local and shared steps, where k is the point contention during the operation's execution interval.

VI. Discussion

This paper presents long-lived algorithms, which adapt to point contention using only read and write operations, for Gather&f, ECollect&f, atomic snapshot and immediate snapshot. The step complexity of the algorithms is adaptive even when local steps are counted and the bounds are very close to the best known (not necessarily adaptive) algorithms for these important problems.

Acknowledgements: We would like to thank Vita Bortnikov for helpful comments.

Algorithm 7 Fully adaptive immediate snapshot [4]: code for process p_i.

Shared:
AS : active set object
Suggest$[1 \ldots n][\infty]$: Array of Collect&$union$ objects
S : FA snaphot // from Algorithm 6
view$[0, \ldots]$: a view for each floor
flag$[0, \ldots][0, \ldots, n-1]$: a flag for each process and each floor

$\langle \Sigma, view \rangle$ procedure px-upscan(*count*: integer)
2: *S*.update(*count*) // store the new value
3: $\langle \Sigma, V \rangle$ = *S*.scan()
4: $\langle tmp, T \rangle$ = *AS*.getSet()
denote $c_{p'}$ = *Suggest*$[p'][V[p'].seq]$
// current collect object for p'
5: for every process $p' \in T$ do
6: $T' = c_{p'}$.collect&$union$()
7: $\forall \langle \Sigma', V' \rangle \in T', \Sigma' > \Sigma$ then $c_{p'}$.put&$union(\langle \Sigma, V \rangle)$
8: *T* = c_p.collect&$union$()
9: $\langle \Sigma, V \rangle$ = element in *T* with minimal Σ part
10: return ($\langle \Sigma, V \rangle$)

view procedure im-upscan(*count*: integer)
11: *AS*.joinSet() // sign-in
12: $\langle f, V \rangle$ = px-upscan(*count*)
13: *view*$[f] = V$
14: $\langle$*tmp*, *U*$\rangle$ = *AS*.getSet()
15: *start_level*$= |V \cup U| + 1$
// estimate the number of participants in lower floors
16: while (**true**) do
// descend through floors $f-1, f-2, \ldots$
17: $f = f - 1$
18: update$_f(p_i)$
19: *curr_level* $=$ |collect$_f()$|
20: *flag*$[f][p_i] = ($*view*$[f] \neq \perp)$
21: *W*$=$ 1s-immss$_f$(*count*, *start_level* $+$ *curr_level*)
22: if ($count >$ *view*$[f][p_i]$) and for some $p_j \in W$, *flag*$[f][p_j] =$ **true** then
23: *AS*.leaveSet() // sign-out
24: return (union(*W*,*view*[*f*]))

REFERENCES

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[2] Yehuda Afek, Hagit Attiya, Arie Fouren, Gideon Stupp, and Dan Touitou. Long-lived renaming made adaptive. In *Proc. 18th ACM Symp. Principles of Dist. Comp.*, pages 91–103, New-York, 1999. ACM Press.

[3] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived and adaptive collect with applications. In *Proc. 40th IEEE Symp. Foundations of Comp. Sci.*, pages 262–272, Phoenix, 1999. IEEE Computer Society Press.

[4] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot. In *Proc. 19th ACM Symp. Principles of Dist. Comp.*, pages 71–80, New-York, 2000. ACM Press.

[5] James Anderson. Composite registers. *Dist. Comp.*, 6(3):141–154, April 1993.

[6] James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proc. 2nd ACM Symp. Parallel Algorithms and Architectures*, pages 340–349, 1990.

[7] Hagit Attiya and Arie Fouren. Polynomial and adaptive long-lived (2k-1)-renaming. In *Proc. 14th Int. Symp. on Dist. Computing*, pages 149–163, Berlin, 2000. Springer-Verlag.

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

[9] Hagit Attiya, Arie Fouren, and Eli Gafni. An adaptive collect algorithm with applications. *Dist. Comp.*, 15(2):87–96, 2002.

[10] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proc. 25th ACM Symp. Theory of Comp.*, pages 91–100, New-York, 1993. ACM Press.

[11] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *Proc. 12th ACM Symp. Principles of Dist. Comp.*, pages 41–52, 1993.

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

[13] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

[14] Michiko Inoue, Shinya Umetani, Toshimitsu Masuzawa, and Hideo Fujiwara. Adaptive long-lived $O(k^2)$-renaming with $O(k^2)$ steps. In *Proc. 15th Int. Symp. on Dist. Computing*, pages 123–135, Berlin, 2001. Springer-Verlag.

APPENDIX

Remark: This appendix reviews material from [8], [14] to help in the verification of our algorithms; it can be ignored.

Our Gather&f algorithm uses the sieve object of Attiya and Fouren [8]. For completeness, we present the pseudo-code of the sieve object (Algorithm 8) and the main lemmas required for our algorithm.

Lemma 1.1 ([8, Lemma 3.2]): If process p_i is inside copy *c* of sieve *s* then all the candidates leave the smaller copies, $1, \ldots, c-1$, of sieve *s*.

Lemma 1.2 ([8, Lemma 3.8]): If some processes access copy c of sieve s, then at least one of them is a winner.

Corollary 1.3 ([8, Corollary 3.11]): The step complexity of accessing the sieve is $O(k \cdot logk)$.

Lemma 1.4 ([8, Lemma 4.2]): Assume that process p_i with interval β_i accesses sieve s and does not win. Then there is a prefix β' of $\beta_i|_1\beta_i|_2 \ldots \beta_i|_s$, such that $|W_s(\beta')| + |A_s(\beta')| \geq s+1$.

The step complexity of the sieve object of [8] is dominated by the latticeAgreement procedure. Inoue et al. [14] reduced both the step complexity and the space complexity of the sieve object, by replacing the latticeAgreement and candidates procedures. Their improved sieve object has $O(k)$ step complexity.

Processes which concurrently enter the same copy obtain the identical set *W* if they obtain non-empty set. To achieve this, procedures latticeAgreement and candidates are used. Each process entering a copy invokes latticeAgreement to capture a snapshot of processes which have entered the same copy in the same round. Then the process invokes candidates to find the minimum snapshot among the snapshots obtained by latticeAgreement. Then procedure candidates can return the minimum snapshot only when the minimum snapshot can be identified, where "minimum" means the minimum one among snapshots obtained in the same copy in the same round including snapshots obtained by other processes later. Since the minimum snapshot is unique, some processes can obtain the identical non-empty set *W*. This can be achieved by invoking collect twice. Then, processes find the minimum snapshot by invoking candidates if the minimum snapshot can be identified. Though their algorithm is simple, it guarantees that at least one processes obtain the non-empty identical set. Moreover, their improved procedures use $O(k)$ registers if k processes enter the same copy in the same round. Therefore, $O(k)$ steps are sufficient to initialize the copy.

Since it suffices to get non-empty set of process identifiers, there is no needed that all the processes will be registered. This allows to simplify the register procedure of Attiya, Fouren and Gafni [9]. They use only one direction of a splitter, and use a collect list instead of a collect tree. The modified splitter returns **stop**, **next** or **abort** instead of **stop**, **left** or **right**, respectively. The properties of the splitter imply that (1) at least one process registers at some node in the collect list, and (2) at most one process registers at each splitter. In collect, a process just searches the list from its root until it reaches an unmarked splitter. The collect returns a view consisting of all process identifiers which are registered before invoking the collect and some process identifiers which register concurrently with the execution of collect.

Algorithm 8 The sieve: code for process p_i.

```
data types:
    processID: int 1..n                          // process's id
    view : vector of ⟨ID, INFO⟩
local variables:
    V, W : view

view procedure enter(s, c, info) // enter (s, c) with info
1:  s.inside[c] = true
2:  V = s.latticeAgreement[c](info)
3:  s.R[c][id_i] = V                  // save the obtained view
4:  W = candidates(s, c)    // get the set of candidates
5:  return(W)                  // return the set of candidates

void procedure exit(s, c)      // leave copy c of the sieve
1:  W = candidates(s, c)    // get the set of candidates
2:  if (⟨id_i, *⟩ ∈ W) then s.count = c + 1
                    // p_i finds it is the winner in (s, c)
3:  s.done[c][id_i] = true                    // p_i is done
4:  w = candidates(s, c)
5:  if (W ≠ ∅ and ∀⟨id_j, *⟩ ∈ W,
        s.done[c][id_i] == true) then
6:      s.allDone[c] = true      // all candidates are done

boolean procedure open(s, c)      // is copy(s, c) open ?
1:  return (s.allDone[c − 1] and not s.inside[c])

view procedure candidates(s, c)
1:  V = s.R[c][id_i]
2:  W = min{s.R[c][id_j] | ⟨id_j, *⟩ ∈ V and
            s.R[c][id_j] ≠ ∅}        // min by containment
3:  if ∀⟨id_j, *⟩ ∈ W, s.R[c][id_j] ⊃ W then return(W)
4:  else return(∅)
```