Space and Step Complexity Efficient Adaptive Collect

Yehuda Afek and Yaron De Levie

School of Computer Science, Tel-Aviv University, Israel 69978

Abstract. Space and step complexity efficient deterministic adaptive to total contention collect algorithms are presented. One of them has an optimal $O(k)$ step and $O(n)$ space complexities, but restrict the processes identifiers size to $O(n)$. Where n is the total number of processes in the system and k is the total contention, the total number of processes active during the execution. Unrestricting the name space increases the space complexity to $O(n^2)$ leaving the step complexity at $O(k)$. To date all deterministic adaptive collect algorithms that we are aware of are either nearly quadratic in their step complexity or their memory overhead is exponential in n.

1 Introduction

In most asynchronous read/write shared memory algorithms each of n processes has its own dedicated register into which it writes new information it has. To collect the information written by others a process reads all the other registers. Obviously, this implementation of a collect operation is wait-free but not adaptive. An implementation of a high level operation is adaptive if the step complexity of the operation is a function of the actual number of processes active rather than n the total number of processes in the system. Three measures of the number of active processes have been defined [AAF+99], *total contention*, *interval contention*, and *point contention*. In an algorithm that is adaptive to *total contention* the step complexity of a high level operation is a function only of the *total number of different processes that have been active in the algorithm execution before this operation terminates*[1].

Following [AKWW] the focus of this paper are adaptive to total contention collect algorithms that are efficient both in space and step complexities. To date all the deterministic adaptive collect algorithms that we are aware of [AKWW,AFG02,MA95,AF03,AST99] are either exponential in space complexity or nearly quadratic in the step complexity (see Table 1).

[1] In *interval contention* the step complexity of a high level operation is a function only of the *total number of different processes that are active during the specific operation sub-execution interval*. In *point contention* the step complexity of a high level operation is a function only of the *maximum number of different processes that are simultaneously active at some point during the operation sub-execution interval*.

In this paper *deterministic* collect algorithms are presented. The first is an $O(n^3)$ space $O(k)$ adaptive step complexity algorithm (§ 4.1). Where n is the total number of processes in the system, k is the total contention of an operation and $\{1 \dots N\}$ is the processes identifiers name space. The second algorithm (§ 4.2) reduces the space complexity to $O(n)$ for the price of increasing the step complexity to $O(k \log\log k)$ and restricting the identifiers size of processes to $O(n)$ (i.e., $N = O(n)$). The $O(k)$ steps $O(n)$ space algorithm is an extension of the second algorithm (§ 4.3). Finally we remove the restriction on the name space from the third algorithm and obtain an $O(n^2)$ space and $O(k)$ step complexity unrestricted name space collect algorithm (also in § 4.3).

Table 1. Results summary. Section 5 algorithms are slightly less efficient than Subsection 4.3 algorithm but much simpler.

Ref & assumptions	Step complexity		Space complexity
	1-st Store by a process	Collect	
[AFG02] *Unrestricted*	$O(k)$	$O(k)$	$O(2^n)$
[AKWW] *Unrestricted*	$O(k/\delta)$	$O(k^2/\delta \log n)$	$O(n^{2+\delta})$
Subsection 4.3 $N = O(n)$	$O(\min(k, n^{3/8}))$	$O(k)$	$O(n)$
Subsection 4.3 *Unrestricted*	$O(k)$	$O(k)$	$O(n^2)$
Section 5 $N = O(n)$	$O(\min(k, \sqrt{n}))$	*if* $(k \leq n^{1/3})$ *then* $O(k)$ *else* $O(\min(n, k \log k))$	$O(n)$
Section 5 *Unrestricted*	$O(k)$	*if* $(k \leq n^{2/3})$ *then* $O(k)$ *else* $O(k \log k)$	$O(n^2)$
[AKWW] *Randomized* $N = O(n)$	$O(k)$	$O(k)$	$O(n^{1.5})$

In [AKWW], Attiya et al. provided a *randomized* $O(n^{1.5})$ space collect algorithm whose expected step complexity is $O(k)$ while assuming that N is of size $O(n)$. (This assumption could be relaxed also in this *randomized* algorithm by using the Moir-Anderson [MA95,AF98] renaming algorithm, resulting in a $O(n^2)$ space complexity and worst case step complexity of $O(k^4)$ (instead of $O(k^2)$), but remaining with linear expected step complexity.)

In [AFK04] Attiya, Fich and Kaplan prove that any $O(f(k))$-adaptive to total contention one-shot collect algorithm requires at least $\Omega(f^{-1}(n))$ multi-writer registers. That is, if the adaptive step complexity is $O(k)$ then the algorithm requires at least $\Omega(n)$ multi-writer registers. Hence, the deterministic algorithms presented in [AKWW,AFG02] still leave a large gap between the corresponding space complexity upper and lower bounds. In this paper we close the gap when

the identifiers size of processes is restricted to $O(n)$ ($N = O(n)$), and the gap has been substantially reduced when N is unrestricted.

In [AF03,AST99] collect algorithms that adapt to point contention are given. Being point contention adaptive these algorithms assume that the register size is $O(n)$ times larger than the register size in the algorithms discussed above.

All algorithms in this paper use a new collect building block, called *Telescopic Watermark Collect* (*TWC*), that has the following two properties: $O(N^{1.5})$ space complexity, and $O(k)$ step complexity if the identifier size of an incoming process is at most quadratic in the number of processes that have accessed the building block so far. To derive from this collect object the first algorithm, which is an $O(n^3)$ space, $O(k)$ steps collect, processes in their first *store* operation go through an adaptive (also in the output name space) MA-renaming $(n \times n)$-matrix (see Figure 2(b))). Then, the new *id* a process obtains from the MA-renaming is used as the *id* with which the process performs a *store* operation in a TWC in which $N_{TWC} = n^2$ (i.e., of (n^3) space). Notice that in this algorithm there is no restriction on the identifiers size of participating processes.

To reduce the space complexity of this algorithm we use a smaller size TWC object and a smaller size MA-renaming matrix. However, such smaller structures may not have room for all n processes. That is, when k the total number of processes that participate in the collect is large enough, processes may fail to obtain a new id from the MA-renaming, or may not fit into the smaller TWC structure. For such processes we either apply recursive divide and conquer or, add a backup structure whose step complexity may be worse but is used only when it is guaranteed that k is above a certain threshold, such as $k > n^{1/4}$ or $n^{1/3}$ and spending that many steps would not hurt the adaptive-linearity of the algorithm.

The model used in the paper is given in Section 2. Preliminary building blocks used in our algorithms are described in Section 3. The linear step linear space algorithm is constructed in Section 4 via a sequence of constructions. In Section 5 we present two additional algorithms with a much simpler presentation and space complexities as efficient as the above, but with a slightly less efficient step complexity ($k \log k$ instead of k). Finally in Section 6 concluding remarks are provided. The code of some of the algorithms appears in the appendices.

2 Model

Following [AFG02,AKWW], here is a brief description of the model. We use the standard asynchronous shared-memory model of computation. A system consists of n processes, $p_1, ..., p_n$, communicating by reading and writing to atomic shared registers. Each process has a unique identifier from the name space $\{1, ..., N\}$.

An algorithm is $f(k)$ adaptive to total contention if there is a non-decreasing function f such that the step complexity of each of its operations is $O(f(k))$, where k is the total number of different processes that have been active since the beginning of the algorithm execution up to the end of the operation execution.

A *collect algorithm* provides two operations *store* and *collect*. A *store(val)* operation by process p_i sets *val* to be the latest value for p_i. A *collect* operation returns a view, a set of values, one for each processor such that, $V[p_i]$ the value for process p_i, is the last value stored by p_i before the *collect* has started or concurrently with the *collect* operation. And the view returned by a *collect* operation C_l that completely follows another *collect* operation C_e is at least as updated as C_e. That is each $V[p_i]$ of C_l is the same or a later value that was written by p_i.

3 Algorithms preliminaries and building blocks

As described in the Introduction each of our algorithms is constructed from a collection of building blocks (structures). In its first *store* operation a process traverses a sequence of these structures until it captures a node in one of the structures and uses the register associated with that node as its dedicated register for the rest of the algorithm execution. In each subsequent invocation of *store* a process updates directly the register it has captured in the first invocation. Therefore, in the following algorithms we mostly describe and analyze the *capture* and *collect* operations. In a *collect* operation a process scans the relevant portions of the data structures that were traversed during the preceding store operations, and collects the information found in all the captured registers.

In all our algorithms, in its first invocation of a *store* operation a process goes (in the *capture* procedure) through several structures and may go through a renaming process several times. Each time taking the previously obtained *id* and generating a newer *id* either by going through a renaming algorithm or by some other arithmetics. In all these algorithms when process p_j starts the first *store* operation it assigns its original *id* into a variable called $name_j$. As it renames itself the *id* in id_j may change while going from one building block to the next, but $name_j$ always holds the original name of the process. In each building block the parameter N denotes the largest *id* that any process may access this building block with. The parameter n denotes the maximum number of processes that may access the object or building block.

A key building block used in all our algorithms is the *Telescopic Watermark Collect* object (TWC). In this section we provide a step wise construction of the Telescopic Watermark object from smaller building blocks, which are: *Watermark Collect* (WC), and *Divided Watermark Collect* (DWC).

Notation:

- A process p_j accesses each of the building blocks WC, DWC, and TWC with with input parameters $name_j$ and id_j. Each of these building blocks takes a parameter N, e.g., $TWC(N)$, which is the largest *id* a process may access this building block with. Copies of the building block are later generated with different values of N.
- Each of the collect algorithms in Sections 4 and 5 is denoted by $(steps, space)-Collect$, indicating this algorithm has *steps* step complexity and *space* space

complexity. E.g., (k, n^2)-Collect denotes an $O(k)$ step $O(n^2)$ space adaptive collect algorithm.

3.1 Watermark Collect(N)

Assume that magically we have an adaptive renaming oracle that renames the processes as they access it with numbers in the range $1, \ldots, f(k)$ where k is the total number of processes that have showed up so far, e.g., $f(k) = 2k - 1$. Then if f is linear, a (n, k)-Collect algorithm is simple to obtain by giving each process a register in an $O(n)$ size array of registers indexed by the new id (call it $index$) a process has obtained. To ensure that a *collect* operation scans only the "occupied" portion of the array we "watermark" the array with a corresponding array of flags. In its first *store* operation a process sets all the flags in the array from the entry that corresponds to the register it has captured in the array until the first flag.

The Watermark Collect is basically that. The most suitable renaming we found is the AF-renaming (Attiya Fouren) *Reflectors Network* - that provides a $(2k - 1)$-renaming in $O(N)$ steps and $O(N^2)$ space [AF98], which are thus the complexities of the Watermark Collect. We append the renaming object with an array A each entry of which consists of three registers $name$, $value$ and $flag$ (see Figure 1(a)). In the *capture* operation (called by the first invocation of the *store* operation) a process p_j uses its id_j to rename itself to obtain a new id, called index. It then captures register $A[index]$ and sets all the flags in $A[index].flag, A[index - 1].flag, \ldots, A[1].flag$ (see the code in figure Code 1). To collect the values stored in the Watermark Collect a process sweeps through the array A from $A[1]$ to the first un-set Flag entry, reading all the values associated with these entries. Clearly the store operation takes $O(N)$ steps (due to the renaming cost), the collect takes $O(k)$ steps and the space complexity is $O(N^2)$ (again due to the Reflectors Network in the AF renaming).

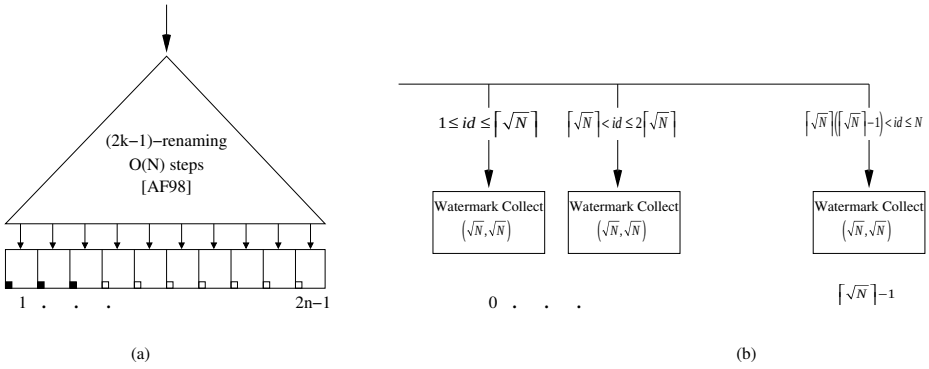

Fig. 1. (a) Watermark Collect(n, N) (b) Divided Watermark Collect(N) constructions.

Notice that replacing the AF-renaming in this construction with any $O(k)$-renaming algorithm with $O(k)$ step complexity and $O(S)$ space complexity results in an $O(k)$ steps collect algorithm with $O(n+S)$ space complexity.

3.2 Divided Watermark Collect(N)

The main problem with the Watermark Collect is the high cost of a *store* operation and of space, due to the AF-renaming. On the other hand, the cost of the *collect* operation is $O(k)$. As a first step in alleviating this issue we reduce the cost of the *store* operation on the account of increasing the cost of the *collect* operation until they are both $O(\sqrt{N})$. At the same time we also reduce the space complexity. This is achieved by using smaller size Watermark Collect objects and assigning processes to the different Watermark Collects according to their ids. Specifically, the Divided Watermark Collect is constructed from a sequence of $\lceil\sqrt{N}\rceil$ Watermark Collects($\lceil\sqrt{N}\rceil$), which results in a $O(\sqrt{N})$ steps store and $O(k+\sqrt{N})$ collect with $O(N^{1.5})$ space.

In its *capture* operation a process p_j such that, $i\lceil\sqrt{N}\rceil < id_j \leq (i+1)\lceil\sqrt{N}\rceil$ invokes the *capture* method of the i'th Watermark Collect object with input parameters ($name_j$, id_j *mod* $\lceil\sqrt{N}\rceil$) (see Figure 1(b) and the code in figure Code 2). To perform a *collect*, a process collects the values from all the $\lceil\sqrt{N}\rceil$ Watermark Collects as was described above.

3.3 Telescopic Watermark Collect(N)

To turn the non-adaptive Divided Watermark Collect into an adaptive collect we assume that the *id*s with which processors enter the object are adaptive. That is we assume that magically each process arrives with an *id* in the range $1,\ldots,f(k)$ where k is the total number of processes that have showed up so far, e.g., $f(k)=k^2$ (if MA-renaming is used). We then construct a sequence of increasing in size Divided Watermark Collect objects and send processes with small *id*s to small DWC objects. In this way the step complexity of the *store* operation is a function of the complexity of the DWC that is used. For example, if a process with $id=t$ accesses a DWC($O(t)$) then its store step complexity (in the corresponding DWC) is $O(\sqrt{t})$. Since $t=O(k^2)$ (assuming $f(k)=k^2$) the resulting step complexity is $O(k)$. To collect the values from this sequence of DWC objects structure we associate with each DWC a flag (similar to the flags in the Watermark object). The flag of a DWC is set if any process stored a value to any DWC of equal or larger size. In the *collect* operation a process collects the values from all the DWCs whose flag is set (until the first one whose flag is not set).

That is, the Telescopic Watermark Collect object consists of a telescopic sequence of increasing in size DWC collect objects, each twice the size of the previous one. The first one is for constant name space C and the last is for name space N. I.e., there are $\approx \log N$ DWC's(see Figure 2(a)). Finally, another array, called sweep array, of size $\approx \log N$, is used to hold the flag registers associated with each DWC.

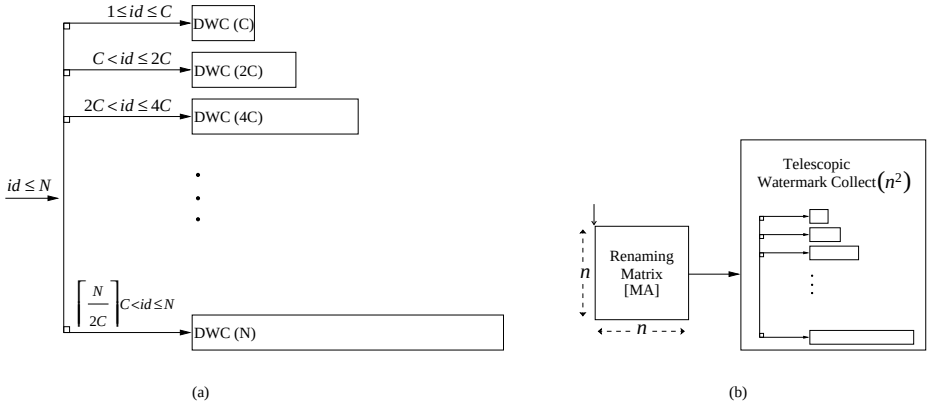

Fig. 2. (a) Telescopic Watermark Collect(N) (b) Unrestricted Name Space (k, n^3)-Collect constructions.

In its *capture* operation (called by the first *store* operation) a process uses its *id* to decide into which DWC it stores its value. A process with $id \leq C$ selects the first one, where as with $2^{i-1}C < id \leq 2^iC$ selects the i'th DWC (see Figure 2(a) and the code in figure Code 3). The process also sets the flags in the Sweep Array according to the DWC it has selected. To perform a *collect*, a process collects from the smallest DWC until the first DWC whose flag is not set. On each such DWC the *collect* will be as described in subsection 3.2.

The Telescopic Watermark Collect thus consumes $O(N^{1.5})$ space and provides $O(k)$ step complexity if the size of the *id* of any incoming process is at most quadratic in the number of processes that have accessed this building block before this process finishes. The space complexity is dominated by the space requirement of the largest DWC(N), which is $\sqrt{N}$ Watermark collects each consuming (by the AF-renaming in it) $O(N)$ space.

Lemma 1. *The step complexity of the* Telescopic Watermark *store operation is* $O(\min(k, \sqrt{N}))$ *if the size of the id of any incoming process is at most quadratic in* k*.*

Lemma 2. *The* Telescopic Watermark Collect *provides* $O(k)$ *step complexity if the size of the id of any incoming process is at most quadratic in* k*.*

4 Stepwise Construction of Linear Step & Linear Space Collect Algorithm

4.1 Unrestricted Name Space (k, n^3)-Collect

As was described above a straightforward method to construct a collect object from the TWC is to frontend a TWC(n^2) with a Moir Anderson adaptive $O(k^2)$-renaming matrix [AF98,MA95], which takes $O(k)$ steps and $O(n^2)$ space. We

thus derived an $O(k)$ steps collect with $O(n^3)$ space (see the construction in Figure 2(b)).

4.2 Restricted Name Space $(k \log\log k, n)$-Collect

Consider a MA-renaming $(n^{1/4} \times n^{1/4})$ matrix followed by a TWC($n^{1/2}$). As long as $k \leq n^{1/4}$ this construction provides an $O(n)$ space collect structure that requires $O(k)$ steps for both *store* and *collect* operations. If $k > n^{1/4}$ then processes may fail to obtain a name in the MA-renaming in their capture procedure. In this event a special flag associated with this MA-renaming is set and these processes are then recursively divided into $n^{1/4}$ groups each of at most $n^{3/4}$ processes (see Figure 3). I.e., a process with id_j such that, $i\lceil n^{3/4} \rceil < id_j \leq (i+1)\lceil n^{3/4} \rceil$ (where in this case id_j is the *name* with which the process started), enters group i and replaces its id_j with $id_j \ mod \ \lceil n^{3/4} \rceil$. In each group we repeat the above recursively. E.g., in the first level of the recursion, each group is of size at most $n^{3/4}$, and contains MA-renaming $(n^{\frac{3}{4}\cdot\frac{1}{4}} \times n^{\frac{3}{4}\cdot\frac{1}{4}})$ matrix followed by a TWC($n^{\frac{3}{8}}$). Processes in a group in the first level that fail to obtain a name in the MA-renaming $(n^{\frac{3}{16}} \times n^{\frac{3}{16}})$ matrix are then recursively divided into $n^{\frac{3}{4}\cdot\frac{1}{4}}$ groups, each of size $n^{\frac{3}{4}\cdot\frac{3}{4}}$. This divide and conquer step is repeated recursively $\mathrm{O}(\log_{\frac{4}{3}}(\log_q n))$ times, getting at the bottom $O(n)$ groups each capable of containing a constant number $q \geq 2$ of processes. In each such group at the bottom level there is a constant size array, where each process captures the entry indexed by the *id* with which it had entered this level (see the code in figures Code 5 and Code 6).

In a collect operation a process essentially performs a DFS on the flagged portion of the tree of groups and in each such flagged group it performs a collect. I.e., if the group is not at the bottom level it performs the collect on the corresponding TWC of the group, otherwise it performs the collect on the corresponding constant size array. A collect on the array simply requires going through all the entries of the array collecting all the values from the captured entries.

Lemma 3. *At level i there are $n^{1-(3/4)^i}$ groups each capable of containing at most $n^{(3/4)^i}$ processes.*

Proof. A group at level i contains at most $n^{(3/4)^i}$ processes. Since the recursive division into groups is a partition, the sizes of all groups at a certain level sum up to n. Therefore there are $n^{1-(3/4)^i}$ groups at level i □

Observation 1 *At each level, a process may belong only to one group while trying to capture a node.*

Observation 2 *if $k \leq n^{1/4}$ all the processes capture a node in the first group.*

Lemma 4. *The store operation in the first invocation takes $O(\min(k, n^{1/4}))$ steps.*

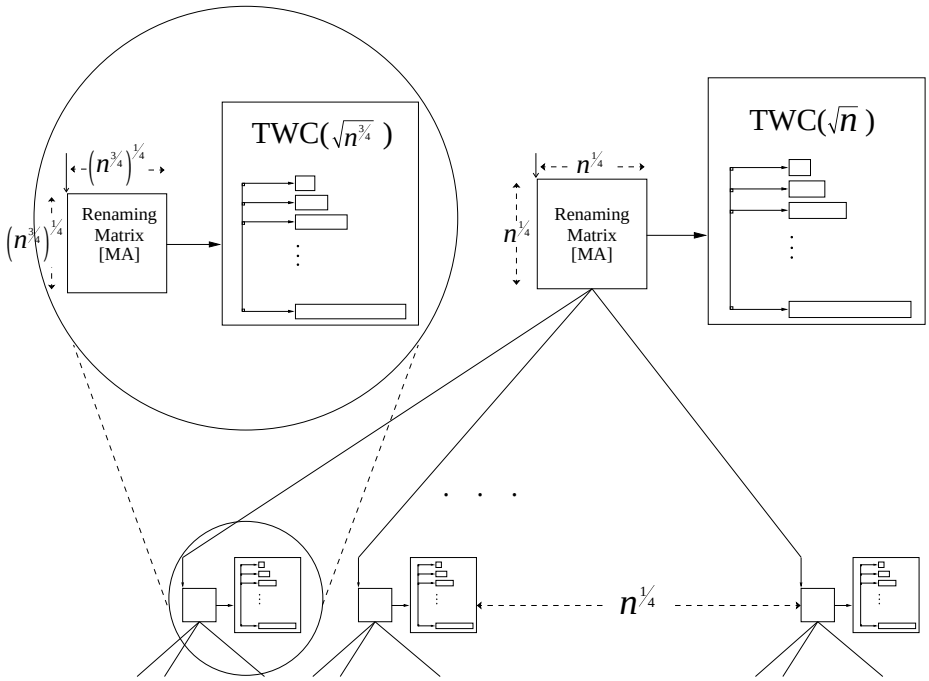

Fig. 3. Restricted Name Space $(k \log\log k, n)$-Collect construction.

Proof. If $k \leq n^{1/4}$ then due to observation 2, any process obtains a name in the MA-renaming of the first group and captures a node in its TWC. Since, obtaining a name takes $O(k)$ steps and capturing a node in the TWC takes $O(k)$ steps as well (Lemma 1), the store operation in this case takes $O(k) = O(\min(k, n^{1/4}))$ steps.

If $k > n^{1/4}$ a process might have captured a node at any level. A process which captured a node at level j ($0 \leq j \leq \log\log n$) went through a series of MA-renaming (one at each level) until either it captures a node at level $j < \log\log n$ in TWC($(n^{(\frac{3}{4})^j})^{\frac{1}{2}}$) or it reaches the bottom level and captures a node in a constant size array. Capturing a node in the bottom level takes constant number of steps. Capturing a node in a TWC($(n^{(\frac{3}{4})^j})^{\frac{1}{2}}$) takes $O(\min(k, \sqrt{(n^{(\frac{3}{4})^j})^{\frac{1}{2}}}))$ (Lemma 1). We are left to prove that going through the series of MA-renaming structures does not take more than $\mathrm{O}(\min(k, n^{1/4}))$ steps.

At level i the MA-renaming matrix is of size $(n^{(\frac{3}{4})^i})^{\frac{1}{4}} \times (n^{(\frac{3}{4})^i})^{\frac{1}{4}}$ therefore, the total number of steps a process spent in all the MA-renamings it had entered is bounded by the summation:

$$\sum_{i=0}^{\log\log(n)-1} (n^{(\frac{3}{4})^i})^{\frac{1}{4}} = O(n^{1/4}) = O(\min(k, n^{1/4}))$$

□

Definition 1. *k_G is the number of processes that enter the MA-renaming associated with group G while trying to capture a node.*

Definition 2. *S_i is the set: $\{G \mid G$ is a group at level i s.t., $k_G > 0\}$.*

Lemma 5. *The collect operation on all the groups in S_i ($i = 1 \ldots \log\log n$), takes $O(k)$ steps.*

Proof. In case level $i < \log\log n$ (not the bottom level), the collect operation on the TWC associated with group $G \in S_i$ takes $\mathrm{O}(k_G)$ steps. This follows Lemma 2 and the fact that the identifiers size of processes that enter the TWC in this group is $\leq {k_G}^2$. Due to Observation 1, $\sum_{G \in S_i} k_G = k$, and therefore, the step complexity of the collect operation on all the groups in S_i is $\mathrm{O}(k)$.

In case $i = \log\log n$ (bottom level), the collect operation on a group takes $\mathrm{O}(1)$ steps. Due to observation 1, there are no more than k groups in S_i. Therefore, the step complexity of the collect operation on all the groups in S_i is $\mathrm{O}(k)$.

Lemma 6. *The total number of groups the collect operation visits is $O(k \log\log n)$.*

Proof. Lets distinguish between two types of groups. First type are visited by both the *collect* operation and some *capture* procedure and the second are those that have been visited only by the *collect* operation. We first claim that at each level there are at most k groups of the first type, and secondly we claim that at each level there are at most k groups of the second type.

The first claim follows from Observation 1. To prove the second claim, observe that each time the collect visits at most $n_l^{\frac{1}{4}}$ groups of type two at level l with the same parent there must have been at least $n_l^{\frac{1}{4}}$ processes that have accessed the parent in their *capture* procedure.

Corollary 1. *The collect operation takes $O(k)$ steps if $k \leq n^{1/4}$ (Observation 2), and $O(k \log\log n) = O(k \log\log k)$ otherwise.*

Lemma 7. *The space complexity is $O(n)$.*

Proof. Each group G at level i (except the bottom level) contains two data structures, MA-Renaming and TWC. The MA-Renaming is a $(n^{(\frac{3}{4})^i})^{\frac{1}{4}} \times (n^{(\frac{3}{4})^i})^{\frac{1}{4}}$ matrix and therefore consumes $\mathrm{O}(n^{(\frac{3}{4})^i})^{\frac{1}{2}})$ space. The TWC is for processes with identifiers of size at most $(n^{(\frac{3}{4})^i})^{\frac{1}{2}}$ and therefore consumes $\mathrm{O}(((n^{(\frac{3}{4})^i})^{\frac{1}{2}})^{1.5}) = \mathrm{O}(n^{(\frac{3}{4})^{i+1}})$ space. Hence, the space required by each group at level i is dominated by the TWC and is thus $\mathrm{O}(n^{(\frac{3}{4})^{i+1}})$.

At level i there are $n^{1-(\frac{3}{4})^i}$ groups (lemma 3). Therefore, in each level except the bottom level, the space complexity of all the groups together is

$$O(n^{(\frac{3}{4})^{i+1}}) \cdot n^{1-(\frac{3}{4})^i} = O(n^{1-\frac{1}{4}\cdot(\frac{3}{4})^i}).$$

Following Lemma 3 there are $\log_{\frac{4}{3}}(\log_q n)$ levels where q is the size of the group at the bottom level which is ≥ 2. Therefore the total space complexity of the construction in all the levels except the bottom one is

$$O(\sum_{i=0}^{\log\log(n)-1} n^{1-\frac{1}{4}\cdot(\frac{3}{4})^i}). \tag{1}$$

We claim that the above summation is $< 15n$

Clearly the space required by the groups at the bottom level is $\mathrm{O}(n)$ □

Proof. of claim:

We bound summation 1 by a geometric series whose sum $< 15n$. The last value in the summation is the largest value and clearly $< n$. The ratio between the $i+1$'st term and the i'th term is:

$$n^{(1-\frac{1}{4}\cdot(\frac{3}{4})^{i+1})-(1-\frac{1}{4}\cdot(\frac{3}{4})^i)} = n^{\frac{1}{4}\cdot(\frac{3}{4})^i-\frac{1}{4}\cdot(\frac{3}{4})^{i+1}} = n^{\frac{1}{16}\cdot(\frac{3}{4})^i}.$$

The smallest ratio is when $i = \log\log(n) - 2$:

$$n^{\frac{1}{16}\cdot(\frac{3}{4})^{\log\log(n)-2}} = n^{\frac{1}{9}\cdot(\frac{3}{4})^{\log\log(n)}} = n^{\frac{1}{9\log n}} = 2^{logn(\frac{1}{9\log n})} \geq 2^{1/9}.$$

Therefore, we can bound the summation (the left side of 1) by

$$n\sum_{i=0}^{\log\log(n)-1}(\frac{1}{2})^{\frac{i}{9}} = n(\frac{1-(\frac{1}{2})^{\frac{\log\log n}{9}}}{1-(\frac{1}{2})^{\frac{1}{9}}}) < 15n$$

□

4.3 Restricted Name Space (k, n)-Collect

In the last algorithm we achieved $(k \log\log k, n)$-collect assuming a restricted name space ($N = O(n)$). Whereas the store step complexity of that algorithm is already $O(k)$, the step complexity of the collect is $O(k \log\log k)$. The complexity of the collect would have been linear in k if (1) in each visited group G the number of steps spend is linear in the number of processes ${k'}_G$ that have entered G's TWC in the *capture* procedure, and (2) the number of groups visited by a collect is $O(k)$. Neither (1) nor (2) are satisfied in the $(k \log\log k, n)$-collect described above, because:

1. If more than $n_G^{1/4}$ processes entered group G, then it is possible that very few processes entered TWC_G, but each with a relatively large identifier. E.g., it is possible that only one process enter TWC_G but with identifier size $O(n_G^{1/2})$. Where TWC_G is the TWC in group G, ${k'}_G$ is the total number of processes which have entered TWC_G in the *capture* procedure, and n_G is the maximum number of processes which may enter group G.
2. A *collect* operation that visits group G might go down to the next level and visit $n_G^{1/4}$ descendent groups even though only $\theta(n_G^{1/4})$ processes entered group G. These $\theta(n_G^{1/4})$ processes might go down to the next level, split equally between $(n_G^{1/16})$ groups and go down another level, causing the *collect* operation to visit again ${n_G}^{1/4}$ descendent groups also in that level. This scenario might repeat itself L times until the bottom level causing the *collect* to visit $L \cdot {n_G}^{1/4}$ descendent groups of G. Hence, if $L \approx \log\log n$ and $k = O(n_G^{1/4})$ then the collect might visit $O(k \log\log k)$ groups.

To deal with the first problem we ensure that any process which captures a node in a TWC_G has identifier of size at most $O({k'}_G^2)$. The first $n_G^{1/4} \times n_G^{1/4}$ MA-renaming matrix does not provide this, since it is possible that more than $n_G^{1/4}$ processes reach this MA-renaming but only a few of them obtain a new name. And their new name sizes may be in the order of $n_G^{1/2}$. To this end, a second $n_G^{1/4} \times n_G^{1/4}$ MA-renaming matrix is inserted before the TWC_G such that each process that obtained a name in the first $n_G^{1/4} \times n_G^{1/4}$ MA-renaming matrix goes through this second renaming matrix of the same size (see Figure 4). Lets distinguish between two cases, ${k'}_G \leq n_G^{1/4}$ and ${k'}_G > n_G^{1/4}$. In the former each process obtains a new identifier of size $O({k'}_G^2)$ in the second renaming matrix. In the Latter these processes obtain "good enough" identifiers in the first MA-renaming. Still, a process which did not obtain a name in the second MA-renaming may not enter the TWC with the identifier it got from the first MA-renaming, since another process might have gotten the same id from the second MA-renaming. Therefore, in this case we add to this identifier a $n_G^{1/4}$ offset and enlarge the TWC to a $TWC_G(2n_G^{1/2})$ such that it accommodates this new identifier size. All these changes do not affect the *collect* operation which remain unchanged.

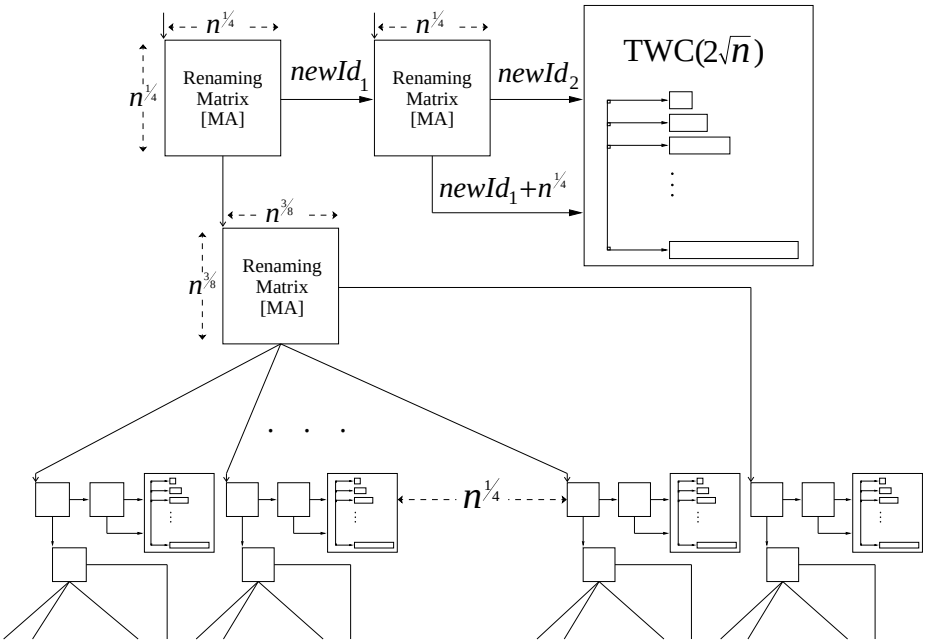

Fig. 4. Restricted Name Space (k, n)-Collect construction. Processes that obtain a new name in a MA-renaming exit the matrix on the right, processes that fail exit at the bottom.

To deal with the second problem we ensure that when a *collect* operation goes down one level from group G it visits only one child group unless at least $\Omega(n_G^{3/8})$ processes entered group G in their *capture* procedure. In the capture procedure at any group G, a process that fails to obtain a name in the first MA-renaming goes through yet another third MA-renaming matrix of size $n_G^{3/8} \times n_G^{3/8}$ (see Figure 4). A process that does not get a new id in the third MA-renaming continues as usual to one of the $n_G^{1/4}$ groups, not before it turns on a flag indicating that these groups are being used (which means that $k_G > n_G^{3/8} \gg n_G^{1/4}$, k_G is the number of processes which visit group G). If a process does get a new id it goes down into a new child group of G, of the same size as the other child groups, and continues in the same way it would have done in any of the other child groups. The only change in the *collect* operation is that when it goes down to the next level its path is divided only if $k_G > n_G^{3/8}$ (instead of $k_G > n_G^{1/4}$).

These two changes improve the step complexity of the collect to $O(k)$ while asymptotically, not affecting the space complexity (see the code in figures Code 7 and Code 8).

Lemma 8. *The* store *operation takes* $O(\min(k, n^{3/8}))$ *steps.*

Proof. If $k \leq n^{1/4}$ then due to observation 2, any process obtains a name in the MA-renaming of the first group and captures a node in its TWC. Since, capturing a node in the TWC takes $O(k)$ steps (Lemma 1) and obtaining a name in both the first and the second $n_G^{1/4} \times n_G^{1/4}$ MA-renaming matrix takes $O(k)$ steps as well, the store operation in this case takes $O(k) = O(\min(k, n^{1/4}))$ steps.

If $k > n^{1/4}$ a process might have captured a node at any level. A process which capture a node at level j ($0 \leq j \leq \log\log n$) went down the tree of groups through a series of MA-renamings. At level $i < j$ the process went through $(n^{(\frac{3}{4})^i})^{\frac{1}{4}} \times (n^{(\frac{3}{4})^i})^{\frac{1}{4}}$ MA-renaming matrix and $(n^{(\frac{3}{4})^i})^{\frac{3}{8}} \times (n^{(\frac{3}{4})^i})^{\frac{3}{8}}$ MA-renaming matrix. At level $i = j$ either it go through two $(n^{(\frac{3}{4})^j})^{\frac{1}{4}} \times (n^{(\frac{3}{4})^j})^{\frac{1}{4}}$ MA-renaming matrices and capture a node at level $j < \log\log n$ in TWC($(2n^{(\frac{3}{4})^j})^{\frac{1}{2}}$) or it reaches the bottom level and captures a node in a constant size array.

Going through $(n^{(\frac{3}{4})^i})^{\frac{1}{4}}$ MA-renaming matrix takes $O(min(k, (n^{(\frac{3}{4})^i})^{\frac{1}{4}}))$. Hence, Going through the series of $(n^{(\frac{3}{4})^i})^{\frac{1}{4}}$ MA-renaming matrices ($i = 1 \ldots j-1$) takes at most

$$\sum_{i=0}^{\log\log(n)-1} (n^{(\frac{3}{4})^i})^{\frac{1}{4}} = O(n^{1/4}) = O(\min(k, n^{1/4}))$$

steps. Going through two $(n^{(\frac{3}{4})^j})^{\frac{1}{4}} \times (n^{(\frac{3}{4})^j})^{\frac{1}{4}}$ MA-renaming matrices takes $O(\min(k, (n^{(\frac{3}{4})^j})^{\frac{1}{4}}))$ steps, and capturing a node in a TWC($(n^{(\frac{3}{4})^j})^{\frac{1}{2}}$) takes $O(\min(k, \sqrt{(n^{(\frac{3}{4})^j})^{\frac{1}{2}}}))$ steps (Lemma 1). Capturing a node in the bottom level takes constant number of steps. All these steps sum up to $\mathrm{O}(\min(k, n^{1/4}))$ steps.

We are left to prove that going through the series of $(n^{(\frac{3}{4})^i})^{\frac{3}{8}}$ MA-renaming matrices does not take more than $\mathrm{O}(\min(k, n^{3/8}))$ steps. If $k > n^{3/8}$ then it takes

at most

$$\sum_{i=0}^{\log\log(n)-1} (n^{(\frac{3}{4})^i})^{\frac{3}{8}} = O(n^{3/8}) = O(\min(k, n^{3/8}))$$

steps. If $k \leq n^{3/8}$ and $j < 5$ then it takes $O(k) = O(\min(k, n^{3/8}))$ steps. Otherwise it takes at most

$$\sum_{i=0}^{4} k + \sum_{i=5}^{\log\log(n)-1} (n^{(\frac{3}{4})^i})^{\frac{3}{8}} = O(k) + O(n^{1/4}) = O(k) = O(\min(k, n^{3/8}))$$

steps. □

Lemma 9. *The collect operation in a visited TWC_G takes $O(k'_G)$ steps if $k'_G > 0$, $O(1)$ otherwise.*

Proof. This follows Lemma 2 and the fact that the identifiers size of processes that enter the TWC in this group is $\leq {k'_G}^2$.

Lemma 10. *The total number of groups the collect operation visits is $O(k)$.*

Proof. The lemma is proved by showing that each process that performs *capture* (out of the k processes) is responsible for $O(1)$ groups that the *collect* visits.

We assign all the groups visited by a collect operation to the different processes that perform capture. The assignment is such that a capturing process maybe assigned many fractions of groups, however the assignment completely covers all the groups. The lemma is proved by showing that the sum of all fractions assigned to a capturing process is $O(1)$.

1. The root group is assigned to the first process that starts executing the capture procedure.
2. A process which enters Group G is assigned $O(n_G^{-\frac{1}{8}})$ out of the $n_G^{\frac{1}{4}}+1$ groups that G is divided into.

To prove that the assignment completely covers all the groups visited by a *collect* operation we distinguish between two cases. Whether the *collect* visits all the child groups of G, or visits just one child group of G. In the former more than $n_G^{\frac{3}{8}}$ capturing processes entered G covering all the child groups since $n_G^{\frac{3}{8}} \cdot O(n_G^{-\frac{1}{8}}) = O(n_G^{\frac{1}{4}})$. In the latter more than $n_G^{\frac{1}{4}}$ capturing processes entered G covering the child group since $n_G^{\frac{1}{4}} \cdot n_G^{-\frac{1}{8}} = n_G^{\frac{1}{8}} > 1$.

We have covered all the groups that the *collect* operation visits. Now we are left to prove that each process is responsible for a constant number of groups as we argued above.

The first item assigns one process to one group. Therefore, we only have to deal with the second item. A process which gets to level i is responsible for $(n^{(\frac{3}{4})^i})^{-\frac{1}{8}}$ hence, a process is responsible for at most

$$\sum_{i=0}^{\log\log(n)-1} n^{-\frac{1}{8}(\frac{3}{4})^i} \tag{2}$$

groups. We claim that this summation < 30 and therefore each process (out of the k processes) is responsible for $O(1)$ groups into which the *collect* entered. □

Proof. of claim:

We bound summation 2 by a geometric series whose sum < 30. The last value in the summation is the largest value and clearly < 1. The ratio between the $i+1$'st term and the i'th term is:

$$n^{\frac{1}{8}(\frac{3}{4})^i - \frac{1}{8}(\frac{3}{4})^{i+1}} = n^{\frac{1}{32}(\frac{3}{4})^i}$$

The smallest ratio is when $i = \log\log(n) - 2$:

$$n^{\frac{1}{32}(\frac{3}{4})^{\log\log(n)-2}} = n^{\frac{1}{18}(\frac{3}{4})^{\log\log(n)}} = n^{\frac{1}{18\log n}} = 2^{logn(\frac{1}{18\log n})} \geq 2^{1/18}$$

Therefore, we can bound summation 2 by

$$\sum_{i=0}^{\log\log(n)-1} (1/2)^{i/18} = \frac{1-(1/2)^{(\log\log n)/18}}{1-(1/2)^{1/18}} < 30$$

□

Corollary 2. *The collect operation takes $O(k)$.*

Unrestricted case: This algorithm has an unrestricted name space variant by simply sending a process in the capture procedure first through a MA-renaming $(n \times n)$ matrix and using the obtained *id* as the *id* with which it enters the above algorithms (see Code 9 for the first variant). The structure of each algorithm is then initiated for n^2 processes, i.e., replace n with n^2 in the above. This results in (k, n^2)-Collect unrestricted name space algorithm.

5 Simple Linear Space & Nearly Linear Step Collect Algorithm

In this section we provide two considerably simpler algorithms which are built from three basic building blocks, a *Telescopic Watermark Collect(N)* (Section 3.3), an *IDs Tree(N)*, and the *Moir Anderson renaming matrix* as suggested in [MA95,AF98].

The *IDs Tree(N)* data structure (which is used with different values of N) is a balanced binary tree with N leaves. With each internal node we associate a register - Flag. The leaf nodes of the IDs Tree are the nodes that processes attempt to capture in the first invocation of a *store* operation, and with each leaf node a $<$ *value* ; *name* $>$ pair of registers is associated. The leaf nodes of each IDs Tree are arranged in an array indexed by processes *id*'s. To capture a leaf-node in a *store* operation a process accesses that node (without contention, it is the only process with that *id*) and then marks the flags along the path from this leaf-node to the root of the tree. To *collect* the values stored in an IDs Tree a process performs a DFS traversal of the flagged portion of the tree (see the code in figure Code 10). Clearly the *store* operation on the IDs Tree takes $O(\log N)$ steps and the *collect* operation takes $O(min(k\log(N),\ N))$ steps. The space complexity is $O(N)$.

Restricted Name Space ($k \log k,\ n$)-Collect In the *capture* operation (see Figure 5(a) and Code 11) called by the first invocation of the *store* operation a process first tries to obtain a new *id* with the MA adaptive renaming matrix of size $n^{1/3} \times n^{1/3}$. If it obtains a new name then it runs with it in a TWC($n^{2/3}$). Otherwise it runs with its *name*, the original id, in an IDs Tree(n).

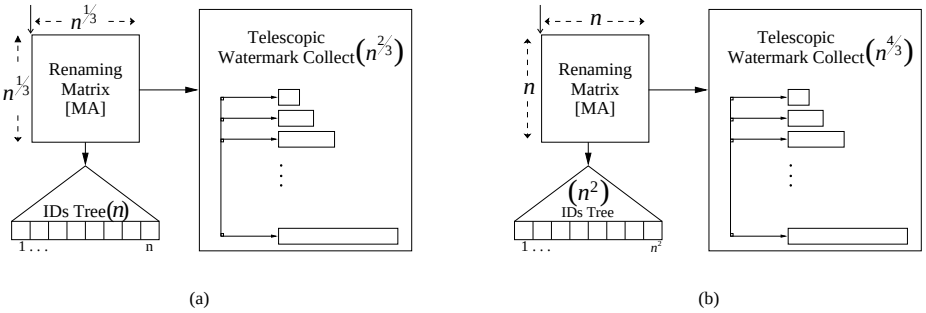

Fig. 5. (a) Restricted Name Space ($k \log k,\ n$)-Collect (b) Unrestricted Name Space ($k \log k,\ n^2$)-Collect constructions.

To perform a *collect*, a process collects from both the TWC($n^{2/3}$) and the IDs Tree(n). Thus a $(min(k \log k, n), n)$-collect algorithm has been obtained.

Unrestricted Name Space ($k \log k,\ n^2$)-Collect An unrestricted name space variant of the above is obtained using the same ideas and following the structure depicted in Figure 5(b).

6 Concluding Remarks

While for processes with identifiers of size $O(n)$, an optimal $O(n)$ space $O(k)$ step complexity collect algorithm has been presented, it is still open to find an algorithm with such complexities for the unrestricted name space case.

A major theme of this paper is the relations between adaptive renaming algorithms and adaptive collect algorithms. Clearly replacing the MA-renaming by a more efficient, in both space and step, renaming algorithm would induce a more efficient, in space, collect algorithm for the unrestricted name space case.

Many renaming and collect algorithms work in the same method as the algorithms in this paper, of capturing a node in a network of nodes [MA95,AF98,AFG02]. In the renaming algorithms the total number of nodes is the name space of the new names, each node represents a new *id*. In the collect on the other hand, the number of nodes relates to the space complexity and does not necessarily effect the other qualities of the algorithm.

The algorithms in this paper combine a linear-adaptive step complexity renaming algorithm which is not optimal in the space of new names (MA-renaming), with an optimal renaming algorithm which is not adaptive in the

step complexity (AF-renaming). By cascading these two algorithms we made the step complexity of the latter effectively adaptive. Still this is not useful for a linear adaptive renaming algorithm, but because in the first store operation of the collect it is only necessary to capture a unique node and not a unique number, this is enough to get a linear collect algorithm.

The adaptive collect algorithms presented in Section 4.3 may be used to construct an adaptive atomic snapshot algorithm by carefully modifying Afek et al. [AAD$^+$93]. The resulting atomic snapshot algorithm runs in $O(k^2)$ steps and $O(n)$ multi-writer registers if the name space is restricted ($N = O(n)$) and with $O(n^2)$ multi-writer registers if the name space is unrestricted.

Acknowledgements: We would like to thank Hagit Attiya and the DISC-05 anonymous referees for helpful comments.

References

[AAD$^+$93] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[AAF$^+$99] Yehuda Afek, Hagit Attiya, Arie Fouren, Gideon Stupp, and Dan Touitou. Long-lived renaming made adaptive. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 91–103, New York, NY, USA, 1999. ACM Press.

[AF98] Hagit Attiya and Arie Fouren. Adaptive wait-free algorithms for lattice agreement and renaming (extended abstract). In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 277–286, New York, NY, USA, 1998. ACM Press.

[AF03] Hagit Attiya and Arie Fouren. Algorithms adapting to point contention. *J. ACM*, 50(4):444–468, 2003.

[AFG02] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

[AFK04] Hagit Attiya, Faith Ellen Fich, and Yaniv Kaplan. Lower bounds for adaptive collect and related objects. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 60–69, New York, NY, USA, 2004. ACM Press.

[AKWW] Hagit Attiya, Fabian Kuhn, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient adaptive collect using randomization. In *Distributed algorithms*, pages 159–173. Also in DISC '04: 18th International Symposium on Distributed Computing.

[AST99] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 262–272, 1999.

[MA95] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, 1995.

APPENDIX

Code 1 Watermark Collect(N)

```
Shared variables:
  A : an array of 2N − 1 entries indexed 1, ..., 2N − 1.
        each entry of which consists of three registers name, value and flag
  AF-Renaming : (2k − 1)-renaming [AF98] for name space N.

node procedure capture(name_j, id_j)
  index = AF-Renaming.getId(id_j)                    // Rename with [AF98].
  acquiredNode = A[index]
  for each i = 1, ..., index                         // Mark the path to the acquired node.
      A[i].Flag = true
  acquiredNode.name = name_j
  return acquiredNode

view procedure collect()
  V = ∅ ; i = 1
  v = A[i]
  while (v.Flag)
      if (v.value ≠ ⊥) then                          // It is a node of a participating process.
          V = V ∪ {⟨v.name, v.value⟩}
      v = A[i++]
  return V
```

A Code

The code of some of the algorithms is provided in this appendix (other codes have been omitted due to space limitations).

The code of all the building blocks is presented as an *object* with two interfaces (methods), *capture* and *collect* i.e., the code of objects does not provide the store operation. The code of the algorithms on the other hand is given for process p_j (and includes the *store* operation).

The *capture* procedure gets a process *name* and *id* as its input and returns a *node* to which this process can store its values.

Code 2 Divided Watermark Collect(N)

Shared variables:
$Watermarks[0, \ldots, \lceil\sqrt{N}\ \rceil - 1]$: an array of Watermark Collect($\lceil\sqrt{N}\ \rceil$).

node procedure **capture**($name_j$, id_j)
let i be s.t. $i\lceil\sqrt{N}\rceil < id_j \leq (i+1)\lceil\sqrt{N}\rceil$
return $Watermarks[i]$.capture($name_j$, $id_j - i\lceil\sqrt{N}\ \rceil$)

view procedure **collect()**
$V = \emptyset$
for $i = 0, \ldots, \lceil\sqrt{N}\ \rceil - 1$ // collects from all the Watermark Collects.
$V = V \cup Watermarks[i]$.collect()
return V

Code 3 Telescopic Watermark Collect(N)

Constants: C
Shared variables:
for $i = 0, \ldots, \lceil\log(N/C)\ \rceil$
DW_i : Divided Watermark Collects(2^iC)
$SweepArray[0, \ldots, \lceil\log(N/C)\ \rceil]$

node procedure **capture**($name_j$, id_j)
if ($id_j \leq C$) then
$acquiredNode = DW_0$.capture($name_j, id_j$)
$SweepArray[0]$ = true
else
let i be s.t. $2^{i-1}C < id_j \leq 2^iC$
for each $k = 1, \ldots, i$ // Mark the path to the Divided Watermark Collect.
$SweepArray[k]$ = true
$acquiredNode = DW_i$.capture($name_j$, id_j)
return $acquiredNode$

view procedure **collect()**
$V = \emptyset$
$i = 0$
while ($i \leq (\lceil\log(N/C)\ \rceil)$ and $SweepArray[i]$)
$V = V \cup DW_i$.collect()
$i++$
return V

Code 4 Simple Collect(N)

Shared variables:
A : an array of N entries indexed $1, \ldots, N$.
each entry of which consists of two registers *name* and *value*

node procedure **capture**($name_j$, id_j)
$acquiredNode = A[id_j]$
$acquiredNode.name = name_j$
return *acquiredNode*

view procedure **collect()**
$V = \emptyset$
for each $i = 1, \ldots, N$
if ($A[i].name \neq \perp$)
$V = V \cup \{\langle A[\text{i}].name, A[\text{i}].value\rangle\}$

Code 5 Recursive Collect1(N)

Constants:
C
Shared variables:
TWC : a Telescopic Watermark Collect($N^{2(1/4)}$).
MA-Renaming : Moir Anderson renaming ($N^{1/4} \times N^{1/4}$)-matrix [MA95].
OverflowFlag : a flag indicating, a process didn't obtain an id in the *MA-Renaming*
if ($N^{3/4} > C$)
COArray = an array of $N^{1/4}$ Recursive Collect1($N^{3/4}$) objects.
else
COArray = an array of $N^{1/4}$ Simple Collect(C) objects

node procedure **capture**($name_j$, id_j)
$newId = MA\text{-}Renaming.getId()$ // Rename with [MA95].
if ($newId \neq \perp$) then
$acquiredNode = TWC$.capture($name_j, newId$)
else // Didn't get an id => the number of participating processes $> n^{1/3}$.
OverflowFlag = true;
let i be s.t. $i\lceil N^{3/4}\rceil < id_j \leq (i+1)\lceil N^{3/4}\rceil$
$id_j = id_j \ mod \ \lceil N^{3/4}\rceil$
$acquiredNode = COArray[i].capture(name_j, id_j)$
return *acquiredNode*

view procedure **collect()**
$V = TWC$.collect()
if (*OverflowFlag*)
for each $i = 1, \ldots, \lceil N^{1/4}\rceil$
$V = V \cup COArray[i].collect()$
return V

Code 6 Restricted Name Space ($k \log\log k,\ n$)-Collect: code for process p_j.

```
Shared variables:
    RC : Recursive Collect1(n) object.
Local variables:                                   // Persistent across invocations of store.
    acquiredNode : The acquired node into which register the processor updates
                   (writes) its new information, initially ⊥.
    name_j : Original unique identifier (∈ {1, ..., n}) of the process.

void procedure store(val)
    if (acquiredNode == ⊥) then                    // Process p_j first invocation of store.
        capture()
    acquiredNode.value = val                       // Write the new information.

void procedure capture()
        acquiredNode = RC.capture(name_j, name_j)

view procedure collect()
    return RC.collect()
```

Code 7 Recursive Collect2(N)

Constants:
C

Shared variables:
TWC : a Telescopic Watermark Collect($2N^{1/2}$).
MA-Renaming1 : Moir Anderson renaming ($N^{1/4} \times N^{1/4}$)-matrix [MA95].
MA-Renaming2 : Moir Anderson renaming ($N^{1/4} \times N^{1/4}$)-matrix [MA95].
MA-Renaming3 : Moir Anderson renaming ($N^{3/8} \times N^{3/8}$)-matrix [MA95].
OverflowFlag1 : a flag indicating, a process didn't obtain an id in the *MA-Renaming1*
OverflowFlag2 : a flag indicating, a process didn't obtain an id in the *MA-Renaming3*
if ($N^{3/4} > C$)
    COArray = an array of $N^{1/4} + 1$ Recursive Collect2($N^{3/4}$) objects.
else
    COArray = an array of $N^{1/4} + 1$ Simple Collect($N^{3/4}$) objects

node procedure **capture**($name_j$, id_j)
    $newId1 = MA\text{-}Renaming1.getId()$ // Rename with [MA95].
    if ($newId1 \neq \perp$) then
        $newId2 = MA\text{-}Renaming2.getId()$ // Rename with [MA95].
        if ($newId2 \neq \perp$) then
            $id_j = newId2$
        else
            $id_j = newId1 + N^{1/2}$
        $acquiredNode = TWC$.capture($name_j, id_j$)
    else // Didn't get an id => the number of participating processes $> n^{1/4}$.
        OverflowFlag1 = true;
        $newId1 = MA\text{-}Renaming3.getId()$ // Rename with [MA95].
        if ($newId1 \neq \perp$) then
            $acquiredNode = COArray[0].capture(name_j, newId1)$
        else
            OverflowFlag2 = true;
            let i be s.t. $i\lceil N^{3/4} \rceil < id_j \leq (i+1)\lceil N^{3/4} \rceil$
            $id_j = id_j \ mod \ \lceil N^{3/4} \rceil$
            $acquiredNode = COArray[i+1].capture(name_j, id_j)$
    return *acquiredNode*

view procedure **collect()**
    $V = TWC$.collect()
    if (*OverflowFlag1*)
        $V = V \cup COArray[0].collect()$
        if (*OverflowFlag2*)
            for each $i = 1, \ldots, \lceil N^{1/4} \rceil$
                $V = V \cup COArray[i].collect()$
    return V

Code 8 Restricted Name Space (k, n)-Collect(): code for process p_j.

```
Shared variables:
    RC : Recursive Collect2( n) object.
Local variables:                                   // Persistent across invocations of store.
    acquiredNode : The acquired node into which register the processor updates
                   (writes) its new information, initially ⊥.
    name_j : Original unique identifier (∈ {1, ..., n}) of the process.

void procedure store(val)
    if (acquiredNode == ⊥) then                    // Process p_j first invocation of store.
        capture()
    acquiredNode.value = val                       // Write the new information.

void procedure capture()
        acquiredNode = RC.capture(name_j, name_j)

view procedure collect()
    return RC.collect()
```

Code 9 Unrestricted Name Space (k, n^2)-Collect(): code for process p_j.

```
Shared variables:
    MA-Renaming : Moir Anderson renaming (n × n)-matrix [MA95].
    RC : Recursive Collect2(n^2) object.
Local variables:                                   // Persistent across invocations of store.
    acquiredNode : The acquired node into which register the processor updates
                   (writes) its new information, initially ⊥.
    name_j : Original unique identifier of the process.
    id_j : Unique identifier (∈ {1, ..., n^2}) obtained by MA-Renaming

void procedure store(val)
    if (acquiredNode == ⊥) then                    // Process p_j first invocation of store.
        capture()
    acquiredNode.value = val                       // Write the new information.

void procedure capture()
    id_j = MA-Renaming.getId()                     // Rename with [MA95].
    acquiredNode = RC.capture(name_j, id_j)

view procedure collect()
    return RC.collect()
```

Code 10 IDs Tree(N)

Shared variables:
$IDsTree$: Balanced binary tree with N leaves; the leaves are indexed $1, \ldots, N$

node procedure **capture**($name_j, id_j$)
$v = IDsTree.leaf[id_j]$ // Capture leaf id_j in IDsTree.
$v.name = name_j$
repeat // Mark the vertices on the path from the captured leaf to the root.
$v.mark$ = true
$v = v.father$
until(v==$\perp$)
return $IDsTree.leaf[id_j]$

view procedure **collect()**
return (DFS($\emptyset$,$IDsTree.root$) // Collect the values in IDs tree

view procedure **DFS**(V: view; v: node)
if ($v \neq \perp$ and $v.mark$) then
if ($v.value \neq \perp$) then // It is a leaf of a participating process.
$V = V \cup \{\langle v.name, v.value \rangle\}$
else
$V = V \cup DFS(V, v.leftChild)$
$V = V \cup DFS(V, v.rightChild)$
return V

Code 11 Restricted Name Space $(k \log k,\ n)$-Collect: code for process p_j.

Shared variables:
TWC : a Telescopic Watermark Collect($n^{2/3}$).
IDsTree : IDs Tree(n).
MA-Renaming : Moir Anderson renaming ($n^{1/3} \times n^{1/3}$)-matrix [MA95].
Local variables: // Persistent across invocations of store.
acquiredNode : The acquired node into which register the processor updates (writes) its new information, initially $\perp$.
$name_j$: Original unique identifier ($\in \{1, \ldots, n\}$) of the process.
id_j : The identifier of the process which could be change by renaming algorithms during the algorithm.

void procedure **store**(*val*)
if (*acquiredNode* == $\perp$) then // Process p_j first invocation of store.
capture()
acquiredNode.value = *val* // Write the new information.

void procedure **capture()** // Capture p_j node.
id_j = *MA-Renaming.getId*() // Rename with [MA95].
if ($id_j == \perp$) then // Didn't get an id => the number of participating processes $> n^{1/3}$.
$id_j = name_j$
acquiredNode = *IDsTree.capture*($name_j, id_j$)
else
acquiredNode = TWC.capture($name_j, id_j$)

view procedure **collect()**
$V = TWC$.collect()
$V = V \cup$ *IDsTree.collect*()
return V