The Concurrency Hierarchy, and Algorithms for Unbounded Concurrency

(Extended Abstract)

Eli Gafni* Michael Merritt† Gadi Taubenfeld‡

Abstract

We study wait-free computation using (read/write) shared memory under a range of assumptions on the *arrival pattern* of processes. We distinguish first between bounded and infinite arrival patterns, and further distinguish these models by restricting the number of arrivals minus departures, the *concurrency*. Under the condition that no process takes infinitely many steps without terminating, for any finite bound $k > 0$, we show that bounding concurrency reveals a strict hierarchy of computational models: a model in which concurrency is bounded by $k + 1$ is strictly weaker than the model in which concurrency is bounded by k, for all $k \geq 1$. A model in which concurrency is bounded in each run, but no bound holds for all runs, is shown to be weaker than a k-bounded model for any k. The unbounded model is shown to be weaker still—in this model, finite prefixes of runs have bounded concurrency, but runs are admitted for which no finite bound holds over all prefixes. Hence, as the concurrency grows, the set of solvable problems strictly shrinks. Nevertheless, on the positive side, we demonstrate that many interesting problems (collect, snapshot, renaming) are solvable even in the infinite arrival, unbounded concurrency model.

This investigation illuminates relations between notions of wait-free solvability distinguished by arrival pattern, and notions of adaptive, one-shot, and long-lived solvability.

*Computer Science Department University of California, Los Angeles, CA 90095. eli@cs.ucla.edu.

†AT&T Labs, 180 Park Ave., Florham Park, NJ 07932-0971. mischu@research.att.com.

‡The Open University, 16 Klausner st., P.O.B. 39328, Tel-Aviv 61392, Israel, and AT&T Labs. gadi@cs.openu.ac.il.

1 Introduction

1.1 Motivation

In most work on the design of shared memory algorithms, it is assumed that the system size, n, is known *a priori*. (We use the terminology *n-arrival model* if n is the maximum number of processes that may be active in a run.) Solvability conditions have been established for the one-shot case, in which each processes arrives at most once [HS93, HS99], while algorithms for various problems have been found for the long-lived case, in which processes may arrive and depart numerous times [AA+99, AF98, AF99, AF2000, AM99, AR93, AST99, MA95].

Work on adaptive algorithms implicitly precludes the use of the system size n as a parameter in a solution. Yet, the system size n plays a role as to whether a problem is solvable or not, by bounding the *concurrency* of the system, defined here as the number of arrivals minus the number of departures.

For instance, taking a snapshot via embedded double collect [AA+93] is guaranteed to terminate in these models, since eventually the number of new arrivals (which interfere with on-going double collects) must be exhausted, letting at least one double collect succeed. (Indeed, termination of the collect routine itself typically depends upon the bound n.)

What if the number of new arrivals can grow without bound? Is the snapshot problem solvable under this relaxed condition? Or collect?

This paper investigates models in which there is no upper bound on the number of active processes: In the *finite-arrival model*, any finite number of processes may take steps in a run, and in the *infinite-arrival* model, an infinite number of processes may take steps in the same run. (Throughout, we assume that the concurrency in any single state is finite.) We distinguish between the following concurrency levels:

- *k-bounded:* There is a finite bound k on the concurrency over all runs.
- *bounded:* In each run the concurrency is bounded.

- *unbounded:* In each prefix of a run R the concurrency is bounded, but there may be no finite bound over the infinite run R.

The definition of unbounded concurrency only makes sense within the infinite-arrivals model, and the reader should assume the latter unless specifically stated otherwise. (Note that distinguishing between k-bounded, and $(k+1)$-bounded concurrency makes sense in either the finite-arrival or the n-arrival model, for any $n \geq k$, and Theorems 2.2 and 2.3 refer to these cases. Also, finite-arrival automatically implies bounded concurrency.)

Our choice to define the concurrency level of a given algorithm as the number of arrivals minus the number of departures, generally corresponds to the *point contention* investigated in previous work on adaptive algorithms. Weaker definitions of concurrency, only peripherally considered here, are the total number of arrivals until a given point, the total number of operations in which processes have taken steps, or the maximum number of processes that take steps while some operation is active. We call these *process*, *operation*, and *interval* contention, respectively.

We restrict our investigations to algorithms that communicate only via atomic (multi-reader, multi-writer) read/write registers, and which are *wait-free*, that is, they guarantee that every participating process will terminate in a finite number of steps regardless of the (fail-stop) behavior of other processes. For such wait-free algorithms a natural time complexity measure is step complexity, the number of accesses to shared memory required to solve a given task. Algorithms may also be *adaptive*, that is, the step complexity of processes' operations is bounded by a function of the actual (point contention) concurrency encountered by the operation. Alternatively, algorithms may be adaptive to process, operation, or interval contention.

1.2 Summary of results

We begin in Section 2 by demonstrating that restricting concurrency increases computational power, by showing that specific problems can be solved under stringent concurrency assumptions, and not when those assumptions are weakened. This implies an infinite hierarchy of computational models, of which the unbounded concurrency model is the weakest.

We then turn in Section 3 to the design of algorithms for unbounded concurrency, focusing first on simple algorithms and techniques and then using more complex constructions to design adaptive algorithms for such problems as collect, snapshot, and renaming.

We assume the reader is familiar with the definitions of the following problems:

1. The (atomic) snapshot problem is to design an object that supports two types of operations, $scan_i$ and $update_i$, by each process i. Executions of *scans* and *updates* must each be considered to have occurred as atomic events [HW87]. Each *scan* operation returns a (finite) list of data values such that the k'th element is the argument of the last $update_k$ that is serialized before that *scan*.

2. The collect problem [AA+93] is to design an object that supports two types of operations, $collect_i$ and $update_i$, by each process i. These operations are similar to the *scan* and *update* snapshot operations, but with weaker semantics for the *collect* operations. For each data element, a *collect* may return either the value of the last $update_k$ operation linearized before it began, or the value of an $update_k$ operation linearized during the *collect*'s execution interval.

3. In the $f(k)$-renaming problem, each process begins with unique names taken from a set of positive integers, and must chose a unique integer from the interval $[1, f(k)]$, where k is the number of processes that has already started before p has completed choosing its new integer name.

1.3 Related work

Wait-free solvability of tasks when there is no upper bound on the number of participating processes has been investigated [GK98], but in this earlier work no (infinite) run has an infinite number of participating processes. The infinite-arrival model has previously been investigated [MT2000], but in models with communication primitives stronger than read/write (e.g. read/modify/write, test&set), or studying problems such as mutual exclusion that do not admit wait-free solution. Adaptive algorithms were defined in [MT93], where the term *contention sensitive* was used to describe such algorithms.

The renaming problem was first solved for message-passing systems [AB+90], and then for shared memory systems [BD89]. There is a tight lower bound of $2k-1$ for the renaming name space [HS99]. Various solutions for renaming have appeared (e.g. [AA+99, AF98, AF2000, AM99, MA95]). This strict bound on name space establishes a strict hierarchy in the read/write model, as $(2k-1)$-renaming is a problem that can be solved by k or fewer processes, but not by any larger number of processes. The snapshot problem was first considered and solved for message-passing systems [CL85]. Subsequently, several constructions of atomic shared memory snapshot objects have appeared (e.g. [AA+93, AR93, AST99]).

2 The Concurrency Hierarchy

In this section we prove that there is a strict infinite hierarchy among the concurrency models defined in the introduction: unbounded concurrency is less powerful than bounded concurrency; bounded concurrency is less powerful than k-bounded concurrency (for any $k \geq 1$); and $(k+1)$-bounded concurrency is less powerful than k-bounded concurrency (for any $k \geq 1$). This hierarchy is shown schematically in Figure 1.

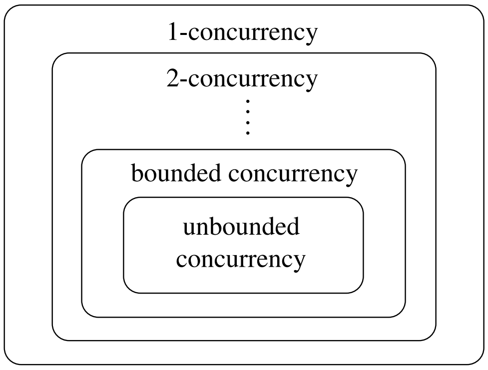

Figure 1: Hierarchy of concurrency models.

The results of this section are essentially a corollary of the following impossibility result (the proof of which uses sophisticated topological characterization of asynchronous computability):

Lemma 2.1 (Herlihy and Shavit [HS99]) *There is no wait-free* $(2n-2)$*-renaming algorithm for* n *processes using read/write shared memory.*

We show that this result implies the following:

Lemma 2.2 *Let* A *be an arbitrary snapshot algorithm for* n *processes, and assume that all the processes try to take a snapshot concurrently. Then, it is possible to schedule them in such a way that each process returns a snapshot which includes all others.*

Proof: Assume to the contrary that there exists a snapshot algorithm, A, for which the lemma does not hold. This implies that we can construct a *splitter* which can be used to divide any set of n processes into two non-empty sets S_1 and S_2. This is done as follows: Each process i simply takes a snapshot in A, and if the size of its snapshot is less than n (i.e., the snapshot does not include all the processes) then $i \in S_1$, otherwise $i \in S_2$. Next we show how using such a splitter we can construct a $(2n-2)$-renaming algorithm, contradicting Lemma 2.1: Let B be an adaptive optimal $2k-1$ renaming algorithm (e.g. in [AF2000]). Let the processes in S_1 use B to choose unique names in the range $1, ..., 2|S_1|-1$, and separately the processes in S_2 use B to choose new names in the range $1, ..., 2|S_2|-1$. Since $(2|S_1|-1)+(2|S_2|-1) = 2n-2$, we can safely replace each name k that was chosen by a process in S_2 with the name $2n-1-k$, and hence assign each one of the n processes a unique name in the range $1,...,2n-2$. ∎

The following corollary follows immediately from Lemma 2.2:

Corollary 2.1 *Let* A *be a snapshot algorithm for unbounded concurrency, let* r *be an arbitrary finite run of* A*, and let* G *be a set of processes which have not taken steps in* r*. It is possible to finitely extend* r *by steps of the processes in* G *so that each of them returns a snapshot which includes all others.*

2.1 Bounded concurrency vs. unbounded concurrency

We first prove that unbounded concurrency is strictly less powerful than bounded concurrency. This is done by showing that there exists a problem that is solvable assuming bounded concurrency, but is not solvable assuming unbounded concurrency.

The *bounded-snapshot* problem is defined as follows: Each process i outputs a set of process id's (a snapshot), S_i. These sets must have the usual snapshot properties: $i \in S_i$, for all S_i and S_j, one is a subset of the other, and if i terminates before j starts then $j \notin S_i$. In addition, take any finite or infinite run in which all participating processes terminate, and reorder the output sets $S_1, S_2, ...$ by set inclusion to get $R_1, R_2, ...$ It is required that for every such sequence R, there exists a bound, say r, such that $|R_1| \leq r$ and for all i, $|R_{i+1}| - |R_i| \leq r$. That is, no snapshot contains more than r "new" elements.

Theorem 2.1 *The bounded-snapshot problem is solvable assuming bounded concurrency, but is not solvable assuming unbounded concurrency.*

Proof: For the bounded case, the algorithm is simply for each process to take a snapshot and return it. (In the next section we show that snapshot is solvable for bounded concurrency.) It is straightforward that in any run in which all participating processes terminate, if we sort the snapshots by size, then the maximum difference between any two successive snapshots is c, where c is the (unknown) bound on concurrency.

Next, we show that it is impossible to solve bounded-snapshot assuming unbounded concurrency. Assume to the contrary that there exists an algorithm, called A, that solves bounded-snapshot assuming unbounded concurrency. We divide the (infinitely many) processes into (infinitely many) disjoint groups G_1, G_2 and so on, such that the size of any group G_i is i, and construct the following infinite run of A: In round 1, the single process in G_1 is activated and it runs alone until it returns a snapshot which includes itself. In round 2, the two processes in G_2 are activated and scheduled in such a way that the two of them return a snapshot which includes all the three processes in $G_1 \cup G_2$. This construction continues similarly, where in round i, all the processes in G_i are activated and scheduled in such a way that all of them return a snapshot which includes all the processes in $\bigcup_{j=1}^{i} G_i$. The fact that such a scheduling is possible follows from Corollary 2.1. Clearly, in this infinite run, there is no bound on the difference between the size of two successive snapshots, as required by the definition of the bounded-snapshot problem, a contradiction. ∎

2.2 k-bounded concurrency vs. $(k+1)$-bounded concurrency

Next we show that for any $k \geq 1$, k-bounded concurrency is strictly more powerful than $(k+1)$-bounded concurrency. This is done by showing that for any $k \geq 1$, there exists a stronger version of the bounded-snapshot problem that is solvable assuming k-bounded concurrency, but is not solvable assuming $(k+1)$-bounded concurrency.

The *k-snapshot* problem is defined as follows: Each process i outputs a set of process id's (a snapshot), S_i. They must have the usual snapshot properties: $i \in S_i$, for all S_i and S_j, one is a subset of the other, and if i terminates before j starts then $j \notin S_i$. In addition, take any finite or infinite run in which all participating processes terminate, and reorder the output sets $S_1, S_2, ...$ by set inclusion to get $R_1, R_2, ...$. It is required that for every such sequence R, $|R_1| \leq k$ and for all i, $|R_{i+1}| - |R_i| \leq k$. That is, no snapshot contains more than k "new" elements.

Theorem 2.2 *The k-snapshot problem is solvable assuming k-bounded concurrency, but is not solvable assuming $(k+1)$-bounded concurrency.*

Proof: For the k-bounded concurrency model, the algorithm is simply for each process to take a snapshot and return it. It easy to see that in any run, if we sort the snapshots by size then the maximum difference between any two successive snapshots is k.

Next, we show that it is impossible to solve k-snapshot assuming $(k+1)$-bounded concurrency. Assume to the contrary that there exists an algorithm, called A, that solves the k-snapshot problem. We construct a finite run of A with $k+1$ active processes: The $k+1$ processes are activated and scheduled in such a way that all of them return a snapshot which includes all $k+1$ active processes. The fact that such a scheduling is possible follows from Lemma 2.2. Clearly, in this finite run, $S_1 = R_1$ and $|R_1| = k+1$. This contradicts the assumption that A solves the k-snapshot problem. ∎

A similar proof shows that a finite hierarchy exists within each n-finite arrivals model:

Theorem 2.3 *For $k < n$, in the n-arrivals model, the k-snapshot problem is solvable assuming k-bounded concurrency, but is not solvable assuming $(k+1)$-bounded concurrency.*

3 Algorithms for Unbounded Concurrency

The results of the previous section establish that unbounded concurrency is the weakest in an infinite hierarchy of computational models. In this section, we demonstrate techniques for designing algorithms for this weakest concurrency model. The first subsection demonstrates some simple techniques and shows that renaming, snapshot, (and hence collect) are solvable in the unbounded concurrency case. Using more complex constructions, subsequent sections show that these problems not only admit solution in the case of unbounded concurrency, but indeed have solutions which are adaptive to the concurrency.

3.1 Simple algorithms and techniques for unbounded concurrency

The results of this subsection establish the solvability of basic tasks in the unbounded concurrency model, and introduce simple techniques that will be applied to more complex constructions in the adaptive algorithms of the next subsection.

3.1.1 Snapshot for unbounded concurrency

There are several snapshot algorithms for the n-arrival model [AA+93, AR93, AST99]. These algorithms do not work for unbounded concurrency. Below, we present a new snapshot algorithm that works in the case of unbounded concurrency (and hence, of course, also for bounded concurrency).

Theorem 3.1 *There is a snapshot algorithm for unbounded concurrency.*

Proof: We first consider the *one-shot* problem, in which each process is assumed to invoke at most one operation. Later we explain how our solution can be trivially made *long-lived*. To simplify the presentation, we consider a version of the snapshot problem where a process returns a list of active process ids, instead of returning a list of values. (Clearly, for the one-shot version a process can then read the value associated with each process in the list.) More formally,the problem is defined as follows: Each process i outputs a set of process id's (snapshot) S_i. They must have the usual snapshot properties: $i \in S_i$, for all S_i and S_j, one is a subset of the other, and if i terminates before j starts then $j \notin S_i$.

The algorithm in Figure 2 implements a snapshot (and operations of process i have $O(max(i^2, k^2)$ step complexity). A description and an informal correctness proof is given after the code. First, process i announces

```
snapshot(i: process_id) - Process i's program.
Shared:
    snap[1..∞]: array of sets, initially all empty
    start[1..∞]: array of boolean, initially all 0
    flag[1..∞]: array of boolean, initially all 0
Local:
    j, k: integer
    collect, doublecollect: set, initially empty

    start[i] := 1
    repeat
    j := 1
    doublecollect := collect
        while flag[j] and i ∉ snap[j] do
                                        /* start collect */
            if start[j] then collect := collect ∪ {j} fi
            j := j + 1
        od                              /* end collect */
/* i sees a snapshot which includes i and uses it */
        if i ∈ snap[j] then doublecollect := snap[j]
    /* i has a snapshot which does not include i */
        else if (i ∉ doublecollect) and
            (doublecollect = collect) then
                snap[i] := doublecollect        /* help */
                for k = i downto 1 do flag[j] := 1 od
                                        /* mark path */
            fi
        fi
                /* doublecollect is the snapshot */
    until i ∈ snap[j] or
        (i ∈ doublecollect and doublecollect = collect)
    return(doublecollect).
```

Figure 2: Snapshot for unbounded concurrency

that it has started by setting $start[i]$ to 1. Then it uses the "double-collect" primitive [AA$^+$93] to take a snapshot of the $start$ bits over the prefix of the array in which the $flag$ bits are set to 1. If this snapshot concludes successfully, but without producing a snapshot that includes i, then this snapshot is posted in $snap[i]$ and i sets $flag[i]$ through $flag[1]$ to 1. Process i then begins another snapshot operation. Since the flags through i are now all set, a second snapshot must include i, and i returns this snapshot if it terminates. Either snapshot operation is interrupted if process i finds itself included in a snapshot recorded in some $snap[j]$, and i returns $snap[j]$.

Proofs similar to those in [AA$^+$93] establish that process i returns legal snapshot values. It remains only to argue that each operation is wait-free: that each snapshot operation terminates.

Suppose process i begins a snapshot operation. Since no $start$ bit is ever changed from 1 to 0, process i cannot loop forever accessing only a finite number of the shared variables–it must either terminate, or eventually read $flag[k] = 1$ for all $k \in \{1, ..., \infty\}$. Suppose the latter occurs.

Let max be the maximum process id associated with snapshot operations that have already begun at the first state satisfying $start[i] = 1$ and $flag[k] = 1$ for all $k \in \{1, ..., i\}$. In this state, $flag[max + 1] = 0$. Later, process i reads $flag[max + 1]$ set to 1. It follows that $flag[max + 1]$ was set by a process p which took a snapshot that started after $start[i]$ and $flag[k]$, for all $k \in \{1, ..., i\}$, were set. This snapshot must include i, and will be read by process i when process i reads $snap[p]$, a contradiction.

This one-shot algorithm is easily made long-lived by reserving infinitely many integer names for each process. Each time a process takes a snapshot, it uses a new unused name. To get the value associated with a process, simply consider the value which is associated with the latest incarnation of that process. ∎

There are two key ideas in this algorithm. The first is that scanning is done *before* extending the flag bits (by setting them to 1). These bits mark a prefix of the array that it is only extended by a process which has successfully completed a snapshot. The prefix can only be extended infinitely if an infinite number of snapshots complete. The output of these scans are available for other operations to borrow, assuring termination. The idea of one operation helping another is common to wait-free algorithms–the twist here is to insist that operations "help first", so that their ensuing updates cannot permanently prevent termination of other operations. This idea guarantees termination in the unbounded case, and is a technique used repeatedly in algorithms below. (For bounded concurrency it would also be fine to do the scanning *after* extending the flag bit, but this can lead to non-terminating executions in the unbounded concurrency case.)

The second key idea is the technique for turning this one-shot algorithm into a long-live one, by assigning each process a new name for each new operation. This technique demonstrates a simple correspondence between the infinite-arrivals model and long-lived versions of many distributed tasks.

3.1.2 One-shot renaming adaptive to process contention

We describe two techniques for modifying adaptive renaming algorithms to unbounded concurrency.

Theorem 3.2 *There is a one-shot renaming algorithm for unbounded concurrency that is adaptive to process contention (and which has linear name space and polynomial time complexity).*

Proof: We use a simple idea, which we call the "interleaving trick", to make specific existing adaptive renaming algorithms work also for unbounded concurrency. In most existing algorithms, a process starts by trying to get number 1 as its new name, if it fails it tries number 2, and so on. However, with unbounded concurrency a process might keep on going and never terminate. In order to ensure termination (without affecting adaptivity), we let the processes compete in the original renaming algorithms only for new *odd* names. However, when a process, say i, first fails to choose number j as its new name where $2i < j$, it simply returns the name $2i$ as its new name.

The interleaving trick can be used with existing one-shot algorithms such as those in [AF98, AM99], resulting in $(4k-2)$-renaming algorithms. ∎

This result takes specific renaming algorithms and modifies them using the interleaving trick to work also for unbounded concurrency.

An alternative technique uses a simple ("black box") transformation transforming any one-shot renaming for finite arrivals into a one-shot renaming that works also for unbounded concurrency. (This technique is an adaptation of ideas used in the adaptive renaming algorithm of Attiya and Fouren [AF99].)

Let A be an arbitrary renaming algorithm for finite arrivals, such as Attiya and Fouren's [AF98], and make infinitely many copies of A, denoted $A_1, A_2, \ldots$ With each A_i, associate a "doorway" bit a_i which is initially set to 0. To get a name, a process p first tries to get a new name by participating in algorithm A_1, if it fails it tries in A_2, and so on. When participating in algorithm A_i ($i \in \{1, 2, \ldots\}$), process p first reads the register a_i. If $a_i \neq 0$ then p fails in A_i, and moves on to participate in A_{i+1}. Otherwise, p sets a_i to 1 and enters A_i. If process p fails to enter any algorithm A_i, ($i \in \{1, 2, \ldots, p-1\}$), it sets a_p to 1 and enters A_p, without first testing the value of a_p.

The doorway bits guarantee that only finitely many processes enter any algorithm A_i, and clearly each process will eventually enter some A_i. Moreover, if only k processes participate, at most k will enter any one A_i, and at most k algorithms will be entered. If each A_i renames k processes to $g(k)$ names, then as in as in [AF99], diagonalizing over the output name ranges of the component A_i results in an output name space of $g(k)^2$.

3.1.3 Snapshot adaptive to operation contention

Using one-shot renaming adaptive to process contention as a pre-processing step before the snapshot algorithm of the previous subsection results in an algorithm adaptive to operation contention.

Corollary 3.1 *There exists a one-shot snapshot algorithm for unbounded concurrency that is adaptive to process contention, and a long-lived snapshot algorithm for unbounded concurrency that is adaptive to operation contention.*

3.1.4 One-shot $(2k-1)$-renaming adaptive to process contention

Theorem 3.3 *There is a one-shot, $(2k-1)$-renaming algorithm for unbounded concurrency that is adaptive to process contention.*

Proof: An optimal name space of size $2k-1$ (where k is the process contention) results by replacing the double collect in the (exponential step complexity) renaming algorithm in [BD89] with the one-shot snapshot algorithm for unbounded concurrency, adaptive to process contention (Corollary 3.1). ∎

3.2 Long-lived adaptive algorithms for unbounded concurrency

In this section, we focus on adaptive algorithms. (Recall, their step complexity is a function to the concurrency, or point contention.) Making use of techniques introduced in the previous section, these algorithms are more complex and generally require very large shared registers and large amounts of local computation. However, they establish that important problems not only lie within the weakest computational model introduced in Section 2, unbounded concurrency, but indeed admit adaptive algorithms.

These algorithms are adaptations of previously known algorithms that were designed for the n-arrival model.

3.2.1 Adaptive renaming for unbounded concurrency

Theorem 3.4 *There is an adaptive renaming algorithm for unbounded concurrency.*

Proof: The required (long-lived) algorithm is a simple modification of the adaptive, n-arrival algorithm by Attiya and Fouren [AF99, AF2000] (and retains its $O(k^3)$

step complexity). This algorithm uses infinitely many data structures called *sieves*, $sieve[1...]$, each with infinitely many copies. Each copy is protected by doorway bits, so in the unbounded concurrency case, only a finite number (of simultaneously active) processes can enter a copy of a sieve. (This is important in part because each sieve has inside it a one-shot adaptive lattice agreement algorithm.)

Their algorithm implements each *getName* operation by successively trying to enter the current copy of $sieve[1, ...]$. They guarantee progress because a process must eventually succeed, by $sieve[n]$ at the latest, in the n-arrival case.

The key insight is that we can use the interleaving trick for process i, if process i fails to enter any of $sieve[1]$ through $sieve[i]$. (Another way to think about it is that there is an extra $sieve[i]$, $sieve[i']$, that only process i ever attempts to enter.) Using this argument, the safety of the scheme is immediate, and only wait-free adaptive termination is in question. But again the "doorway" bound on entrants to any single sieve copy, together with the interleaving rule, make this straightforward. ∎

3.2.2 Adaptive collect for unbounded concurrency

Theorem 3.5 *There is an adaptive collect algorithm for unbounded concurrency.*

Proof: The (long-lived) algorithm is presented in Figure 3. It is an adaptation of the (long-lived) adaptive collect algorithm for n-arrivals by Afek, Touitou, and Stupp [AST99], using the "help first" technique introduced in the previous section. (For simplicity of presentation, this collect algorithm returns tuples containing every update performed by each process–restricting it to return only the most recent update is straightforward. The resulting algorithm retains the $O(k^4)$ step complexity of the original.) Both our and the original Afek, *et al* collect algorithms use an array, $A[Level]$, to store update tuples, which processes access in mutual exclusion by running an adaptive renaming algorithm to pick an entry $Level$. Of course, in our algorithm we use the adaptive renaming algorithm for unbounded concurrency from Theorem 3.4. Both collect algorithms also have an auxiliary array, $C[Level][Pid]$, indexed by $last[Level]$, by which processes "bubble up" collect values towards the top of the array. In the original collect algorithm, each step of this process involves scanning lower levels of the array. Termination was assured by the bound on the number of participating processes, but non-terminating runs are possible in the case of unbounded concurrency.

To apply the "help first" technique, we have added an array $B[Level]$ which each process uses to record it's progress in carrying out successive scans. We also insist that at each step in "bubbling up", a process scan this array from $B[1]$ to its current level, using the result to label the collected values it copies upward. The termination step for the downward scan is now augmented by looking for one's own counter in the values observed, and borrowing the associated collect value in that case.

The correctness proof parallels that of the original algorithm; below we include a detailed proof for the key safety lemma that is affected by our change. ∎

Let e be an execution of the adaptive collect algorithm, and let s be an execution of $scan(x)$ contained in e. Let $process(s)$ be the process executing s, and let $(sis(s), itemset(s))$ be the $(ScanItemSet, ItemSet)$ returned by s. Moreover, let $ref(s)$ be the execution of the encapsulating $refresh()$ operation by $process(s)$, let $input(s)$ be the $ItemSet$ input to $ref(s)$, let $index(s)$ be the name obtained by $process(s)$ in $ref(s)$, and let $sctr(s)$ be the value of $scancounter$ after it is incremented in line 16 in s. Similarly, let g be an execution of $gather(x)$ contained in e. Let $process(g)$ be the process executing g, and let $itemset(g)$ be the $ItemSet$ returned by g. Moreover, let $scan(g)$ and $ref(g)$ be the executions of the corresponding encapsulating $scan$ and $refresh$ operations by $process(g)$, let $input(g)$ be the $ItemSet$ input to $ref(g)$, let $index(g)$ be the name obtained by $process(g)$ in $ref(g)$, and let $sctr(g)$ be the value of $scancounter$ after it is incremented in line 16 in $scan(g)$.

We say a $refresh$ operation by process p *crosses* row k when the statement $last[i] := p$ (line 11) is executed.

Lemma 3.1 *Let e be an execution of the adaptive collect algorithm, and let g and s be executions of $gather(k)$ and $scan(k)$, correspondingly, contained in e. If $ref(s)$ crosses k before the beginning of g, then $input(s) \subseteq itemset(g)$.*

Proof: By induction on the length of e. The $gather(k)$ execution g reads a process id, q', from $last[k]$ (line 22, call this event α), then reads values tag', sis' and set' from $C[k][q']$ (line 23, call this event β), and returns according to one of three conditions:

1. $tag' = 1$
   Then there is an execution s' of $scan(k)$ such that s' precedes β, $ref(s')$ by process q' is the last $refresh$ operation to cross k before α, and s' contains a $gather(k)$ operation g'. Moreover, $itemset(g') \subseteq itemset(s') \subseteq itemset(g)$. If $ref(s)$ crosses k before the beginning of g, then $ref(s)$ crosses k before the beginning of g', and by induction $input(s) \in itemset(g)$.

2. $tag' = 0$ and $(process(g), sctr(g)) \in sis'$.
   Then there is an execution s' of $scan(k + 1)$ by

```
Type:
    Pid = process id, 1, ...
    Level = 1, ...
    Counter = 0, ...
    Item = (Pid, Value, Counter)
    ItemSet = Set of Item
    ScanItem = (pid : Pid, ctr : Counter)
    ScanItemSet = Set of ScanItem
    TaggedItemSet =
        (tag : Boolean, sis : ScanItemSet, itemset : ItemSet)
Shared:
    A[Level]: Array of ItemSet, initially all {}
    B[Level]: Array of ScanItem, initially all (0, 0)
    last[Level]: Array of Pid, initially all 1
    C[Level][Pid]: Array of TaggedItemSet,
        initially all (1, {}, {})
Local:
    scancounter, updatecounter : Counter, initially 0

procedure update(value : Value)
1    updatecounter = updatecounter + 1
2    refresh({(p, value, updatecounter)})

function collect() returns ItemSet
3    s := refresh({})
4    refresh(s)
5    return(s)

function refresh(S : ItemSet) returns ItemSet
6    index := Rename(p)
7    A[index] := A[index] ∪ S
8    s := scan(index + 1)
9    for i = index down to 1 do
10       C[i][p] := (0, s.sis, s.itemset)
11       last[i] := p
12       s := scan(i)
13       C[i][p] := (1, s.sis, s.itemset)
     od
14   ReleaseName(index)
15   return(s)

function scan(k : Level)
        returns (sis : ScanItemSet, itemset : ItemSet)
16   scancounter := scancounter + 1
17   B[index] := (p, scancounter)
18   tmp := {}
19   for i = 1 up to k do
20       tmp := tmp ∪ {B[i]} od
21   return(tmp, gather(k))

function gather(k : Level) returns ItemSet
22   q := last[k]
23   (tag, sis, set) := C[k][q]
24   if (tag = 0) and ¬((p, scancounter) ∈ sis)) then
25       set := set ∪ gather(k + 1) fi
26   return(set ∪ A[k])
```

Figure 3: Update and collect for unbounded concurrency, process p's program

process q' that reads $(process(g), sctr(g))$ from $B[index(g)]$ after the latter is written by $scan(g)$, and such that $set' = itemset(g')$ for an execution g' of $gather(k+1)$ by q' that begins after g begins.

If $ref(s)$ crosses k before the beginning of g, then either $index(s) = k$ or $ref(s)$ crosses $k+1$ before the beginning of g and so before the beginning of g'. In the latter case, by induction $input(s) \subseteq itemset(g') \subseteq itemset(g)$. But if $index(s) = k$, $process(g)$ reads $input(s)$ from $A[k]$ in line 26 and we are done.

3. $tag' = 0$ and $\neg((process(g), sctr(g)) \in sis')$.
Then $item(g)$ is the union of set', $A[k]$ (line 26) and $item(g')$, where g' is the execution of $gather(k+1)$ in the recursive call in line 25.

   As in the previous case, if $ref(s)$ crosses k before the beginning of g, then either $index(s) = k$ or $ref(s)$ crosses $k+1$ before the beginning of g and so before the beginning of g'. In the latter case, by induction $input(s) \subseteq itemset(g') \subseteq itemset(g)$. But if $index(s) = k$, $process(g)$ reads $input(s)$ from $A[k]$ in line 26 and we are done. ∎

Corollary 3.2 *There is an adaptive snapshot algorithm for unbounded concurrency.*

Proof: Afek, Touitou, and Stupp use their adaptive collect algorithm to construct an adaptive snapshot algorithm for n-arrivals [AST99]. The latter uses the "help first" technique, and so, using the adaptive collect from Theorem 3.5, also works for the unbounded concurrency case. (It also retains its $O(k^4)$ step complexity in the unbounded concurrency model.) ∎

4 Discussion

We showed, using somewhat contrived problems, that the unbounded concurrency model is strictly weaker than bounded concurrency. We have demonstrated that interesting problems, such as snapshot and renaming, which are known to have solutions for a finite number of processes (the n-arrivals model), are also solvable assuming unbounded concurrency, even adaptively.

The bounded-snapshot problem, which is used to separate the bounded concurrency from unbounded concurrency, is defined using an existential quantifier for infinite input-output relations. We would like to to find a problem which can be defined (as a finite input-output task) without the need to use such a quantifier. Below we define such a problem, called the multi-snapshot problem, which is solvable assuming bounded concurrency, and which we conjuncture, is not solvable assuming unbounded concurrency.

The *multi-snapshot* problem is defined as the input/output relation resulting when the following algorithm is run in the bounded concurrency model: Each process i has a register that contains a finite set of pairs of the form (i', j), where i' is a process id and j is a natural number. Process i performs $update_i(i,1)$, $s := scan_i$, $update_i(\{(i,2)\} \cup s)$, $s := scan_i$, $update_i(\{(i,3)\} \cup s)$,..., continuing as long as the number of new processes (ignoring the counter) seen in each scan is more than the number of new processes in the last (taking the empty set as the implicit first snapshot). When a snapshot doesn't grow by more than the last, it returns the sequence of snapshots.

It is also of interest to more generally extend the theory on the topological structure of asynchronous computability for finitely many processes from [HS99], to cover also the case of infinitely many processes and different bounds on concurrency.

References

[AA+93] Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., and Shavit, N. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873-890, 1993.

[AA+99] Afek, Y., Attiya, H., Fouren, A., Stupp, G., and Touitou, D. Long-lived renaming made adaptive. In *Proc. 18th Annual ACM Symp. on Principles of Distributed Computing*, 91-103, May 1999.

[AB+90] Attiya, H., Bar-Noy, A., Dolev, D., Koller, D., Peleg, D., and Reischuk, R. Renaming in an asynchronous environment. *Journal of the ACM* 37(3)524-548, July 1990.

[AF98] Attiya, H. and Fouren, A. Adaptive wait-free algorithms for lattice agreement and renaming. in *Proc. 17th Annual ACM Symp. on Principles of Distributed Computing*, 277-286, 1998.

[AF99] Attiya, H. and Fouren, A. Adaptive long-lived renaming with read and write operations. Technion Technical Report 0956, March 1999.

[AF2000] Attiya, H. and Fouren, A. Polynomial and adaptive long-lived $(2k-1)$-renaming. *Proceedings of the 14th International Symposium on Distributed Computing: LNCS, 1914.* pages 149–163. Springer Verlag, Oct. 2000.

[AM99] Afek, Y., and Merritt, M. Adaptive, wait-free $(2k-1)$-renaming. In *Proc. 18th Annual ACM Symp. on Principles of Distributed Computing*, 105-112, May 1999.

[AR93] Attiya, H. and Rachman, O. Atomic snapshots in $O(n \log n)$ operations. In *Proc. 12th Annual ACM Symp. on Principles of Distributed Computing*, 29-40, 1993.

[AST99] Afek, Y., Stupp, G., and Touitou, D. Long-lived and adaptive collect with applications. In *Proc. of 40th IEEE Symp. on Foundations of Computer Science*, Oct. 1999.

[BD89] Bar-Noy, A. and Dolev, D. Shared memory versus message-passing in an asynchronous distributed environment. In *Proc. 8th Annual ACM Symp. on Principles of Distributed Computing*, 307-318, 1989.

[CL85] Chandy K. M., and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computing Systems, 3*, 1 (Jan. 1985) 63–75.

[H91] Herlihy, M. Impossibility results for asynchronous PRAM. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 327–336, July 1991.

[HS93] Herlihy, M., and Shavit, N. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 111–120, May 1993.

[HS99] Herlihy, M., and Shavit, N. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858-923, 1999.

[HW87] Herlihy, M., and Wing, J. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, July 1990.

[GK98] Gafni, E., and Koutsoupias, E. On uniform protocols, 1998 `http://www.cs.ucla.edu/~eli/eli.html`.

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

[MT93] Merritt, M and Taubenfeld, G. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993. (Also published as an AT&T technical memorandum in May 1991.)

[MT2000] Merritt, M. and Taubenfeld, G. Computing with infinitely many processes. *Proceedings of the 14th International Symposium on Distributed Computing: LNCS, 1914.* pages 164–178. Springer Verlag, Oct. 2000.