The Assignment Problem

Carole Delporte[1], Hugues Fauconnier[1], Eli Gafni[2], and Giuliano Losa[2]
[1]University Paris Diderot [2]UCLA

October 27, 2017

Abstract

In the allocation problem, asynchronous processors must partition a set of items so that each processor leave knowing all items exclusively allocated to it. We introduce a new variant of the allocation problem called the assignment problem, in which processors might leave having only partial knowledge of their assigned items. The missing items in a processor assignment must eventually be announced by other processors.

While allocation has consensus power 2, we show that the assignment problem is solvable read-write wait-free when k processors compete for at least $2k-1$ items. Moreover, we propose a long-lived read-write wait-free assignment algorithm which is fair, allocating no more than 2 items per processor, and in which a slow processor may delay the assignment of at most n items, where n is the number of processors.

The assignment problem and its read-write solution may be of practical interest for implementing resource allocators and work queues, which are pervasive concurrent programming patterns, as well as stream-processing systems.

1 Introduction

We consider the problem of uniquely allocating items to processors in an asynchronous shared-memory system. This problem is pervasive in concurrent and distributed programming, such as in work queues, where jobs are to be dispatched to several processors, or in resource allocation systems (e.g. memory allocation, allocation of process descriptors in an operating system, etc.).

Classical examples of resource-allocation problems include the dining-philosophers problem [9], the mutual exclusion problem, and its generalization in the L-exclusion problem [11]. Because some processors may have acquired all the resources while others are trying to acquire resources, these problems trivially admit no wait-free solutions, and they are therefore usually studied under the assumptions that processors are scheduled fairly. For example, Peterson's mutual exclusion algorithm [23] or Lamport's Bakery algorithm [17] solve the mutual exclusion problem under fair scheduling, but may only be as fast as the slowest processor in the system.

The long-lived renaming problem [21] is a relaxed form of allocation in which processors repeatedly acquire and release a single item each taken among a number of items larger than the number of processors. Because of the availability of spare items, there is still possibility for progress even if processors do not release acquired items. Moreover, given enough spare items, the flexibility processors have in choosing their items allows to solve the long-lived renaming problem wait-free and using only atomic registers. However, having spare item may be considered a wasted, as some items are not allocated to any processor. Nevertheless, read-write wait-free algorithms are advantageous: wait-freedom provides the highest degree of fault-tolerance, and atomic registers are implementable from some of the most unreliable communication primitives, e.g. safe registers [18].

We ask whether there is a trade-off between long-lived renaming, which is solvable read-write wait-free but leave some items unallocated, and problems like mutual-exclusion, which allow full allocation but are only implementable read-write under fair scheduling of all processors. To simplify our analysis, we start by considering single-use allocation problems (in which processors try to acquire items only once and do not release them) in the wait-free model.

In the (single-use) allocation problem, a set of items R numbered 1 to r must be allocated to a set of $n = r$ processors such that each item is allocated to a single processor. If all we require is that each processor get at least one item, then the problem is trivial: we can statically pre-allocate items to processors and solve the problem without any synchronization. However, we would still waste items if not all processors participate. We cannot require that the first processors to come take all items, as this would preclude every processor getting at least one item. But we can require that the first processors get the first items, which would help at least for provisioning items (e.g. if one knows that only k processors will request items, no need to provision more than k). Formally, assuming that $r \geq n$, we require that when k processors request items, then exactly the first k items (no more, no less) are allocated. As we will see in Section 3, the single-use allocation problem cannot be solved wait-free with registers, and synchronization primitives such as test-and-set are needed. Can a relaxed allocation problem be solved read-write wait-free? Single-use renaming [5] can, but we have seen that it wastes items.

For applications such as work queues or stream processing, it may be sufficient to give a processor a partial allocation of items, letting this processor work with its partial allocation before coming back to get its remaining allocated items. If items are jobs to be completed, this would allow a processor to start working on a job before knowing the full set of jobs it has been allocated, and to come back later when it finishes its first jobs to retrieve its remaining allocated jobs. If items are resources needed for some task, a processor may start working with restricted resources while waiting for its full resource allocation to be revealed.

To capture the intuition above, we propose the assignment problem. In the single-use version of the assignment problem, each processor p must announce a set of items $D[p]$ and the corresponding assignment $a[p] : D[p] \to P$ describing, for each item $i \in D[p]$, the processor $a[p][i]$ to which i is assigned to. To solve the assignment problem given a non-triviality parameter $f : \mathbb{N} \to \mathbb{N}$ (where f is strictly increasing), four conditions must be met:

Fairness: every processor p announces an assignment in which it gets at least one item.

Consistency: if two processors announce an assignment for item i, then they assign i to the same processor.

Non-Triviality: for every k, if k processors participate, then

1. only the first $f(k)$ items may be assigned, and
2. if all k processors terminate, then every item in $\{1, \ldots, f(k)\}$ is announced.

In formulating the assignment problem, we hope to obtain a problem that is solvable read-write wait-free. However, one can see that this will depend on the non-triviality condition: if the non-triviality condition stipulates that exactly the first k items must be allocated when k processors participate, then the allocation problem and the assignment problem coincide and consensus power 2 will be needed. Hence the question: under what non-triviality condition is the assignment problem solvable read-write wait-free?

We can derive a non-triviality lower bound by observing that the assignment problem offers a solution to the $f(k)$-adaptive-renaming problem [5]. As Gafni et al. [12] have shown, $\left(2k - \lceil \frac{k}{n-1} \rceil\right)$-adaptive-renaming can be used to solve $(n-1)$-set-consensus, and is therefore impossible to solve read-write wait-free. Therefore, $f(k) = 2k - \lceil \frac{k}{n-1} \rceil$ is a lower bound under which the assignment problem is not solvable read-write wait-free. As we will see, this lower bound is tight. Moreover, given this bound, the best we can hope for in terms of fairness of the assignment is that each processor get at most two items. Surprisingly, this is also achievable. The assignment problem and its read-write wait-free solution are presented in Section 4.

Finally, in Section 5, we extend our investigation to a long-lived version of the assignment problem, in which processors repeatedly come back to get new items from an infinite stream of items, and we propose a read-write wait-free solution based on the single-use algorithm. We then present an optimized version of the long-lived algorithm which bounds by a constant the number of items that may be left unassigned because of a slow processor. In a solution based on mutual exclusion, e.g. using the Bakery algorithm, a slow processor can arbitrarily delay the whole system even when the slow processor is not in its critical section. In contrast, the optimized long-lived assignment algorithm presented in Section 5 ensures that a slow processors delays the assignment of at most n items while the other processors suffer no delay.

All the algorithms presented are formalized in the PlusCal [19] language, and their properties in a system of 4 processors have been verified using the TLC model-checker [24]. The full PlusCal formalizations are available at `https://losa.fr/research/assignment`.

2 Model

We consider a set P of asynchronous processors communicating through a single-writer multi-reader shared memory and optionally through tasks and linearizable objects. When P is finite, we write n for the number of processors. Each processor has a private local state and a private read-only input. The memory consists in one register L_p per processor p. A processor can take local steps, read steps, write steps, object-invocation steps, task input steps, and task output steps. The next step of a processor is always enabled, i.e. a processor cannot wait for a condition. In a read-write algorithm, processors can only take read steps and write steps.

A write operation by processor p writes to register L_p only, and a read operation returns an atomic snapshot [2] of the entire shared memory. Objects are sequential state-machines with a transition relation relating pre-state, operation, response, and post-state; an object-invocation step, taking an operation as parameter, changes the object state and returns a response to the invoking processor in a single step and according to the transition relation of the object.

Let an input vector be a partial function from processor to input, and an output vector be a partial function from processor to output. A task is a partial function mapping an input vector to a set of output vectors which have the same domain as the input vector. Informally, given the set of participating processors in a run and their input, a task describes the allowed outputs of those processors. A task input step does not return any response to a processor, while a task output step non-deterministically produces a response such that the output vector of the task, as observed so far, can be completed to an output vector in relation with the input vector of the task, as observed so far.

An algorithm assigns an initial local state and a deterministic sequential program to every processor (subject to the constraint that if a processors takes a task input step, then its next step must be the corresponding task output step). A run of an algorithm consists of an input vector and an infinite sequence of processor steps, where each processor starts with the input assigned to it by the input vector and takes steps according to the algorithm. A processor may terminate by finishing its program and writing an output in its local state, in which case it only takes stuttering steps (i.e. steps that do not change its state) thereafter. An algorithm is wait-free if every processor that takes infinitely many steps eventually terminates.

We say that a processor p participates in a run if p takes at least one step. Throughout the paper, we write Q for the set of participating processors in a run and k for their number ($k = |Q|$).

An algorithm solves a task Δ when

1. the algorithm is safe: in every run in which the input vector is in the domain of Δ and all participating processors terminate, the input vector is related by the task to the output vector observed in the run, and
2. the algorithm is wait-free.

Note that in most of the tasks that we use in this paper, a processor receives no input. In this case, a task reduces to a relation between participating sets and output vector.

The consensus number of a task or object type is the maximum number of processors for which there exists a wait-free algorithm that solves the consensus task using registers and instances of the task or object type. By convention, every task or object has consensus number at least 1. Consensus is impossible to solve with registers even for two processors [20], therefore registers have consensus number 1.

Throughout the paper, we make use of solutions to the following three input-less tasks. In the test-and-set task, exactly one participant must output 1 while all others must output 0. Test-and-set has consensus number 2, therefore it has no read-write wait-free solution. However, test-and-set can be solved using 2-processors consensus.

In the immediate-snapshot task [6], each participating processor p must output a set of participating processors $is[p]$ such that $p \in is[p]$ and, for every two processors p and q, $is[p] \subseteq is[q]$ or $is[q] \subseteq is[p]$, and if $p \in is[q]$ then $is[p] \subseteq is[q]$. The immediate-snapshot task is solvable read-write wait-free.

In the $(2k-1)$-adaptive-renaming task [4], each participating processor p must output a unique integer $name[p]$, called p's name, such that, for every k, when k processors participate, $1 \leq name[p] \leq 2k-1$. The $(2k-1)$-adaptive-renaming task are solvable read-write wait-free.

In a long-lived problem, a processor receives a new input each time it produces an output, and must match the new input with an output. We consider long-lived problems that can be specified as tasks for infinitely many processors, i.e. such that there is a task Δ for infinitely many processors such that a solution to Δ can be transformed into a solution to the long-lived problem by having each processor pick a fresh identifier for itself each time it receives a new input (e.g. by using identifiers of the form $\langle p, i\rangle$ where i is an integer incremented each time a fresh identifier is needed).

Note that this class of long-lived problem excludes problems in which a processor operation is constrained by the operations it performed before. An example of task that is outside the class is the long-lived renaming problem [22], in which a process can release a name only if it previously acquired it. Also note that in a long-lived problem for n processors, the number of concurrent processors is trivially bounded by n. Therefore, since we consider tasks for infinitely many processors only as a model of long-lived problems, we will assume that the number of processors active at any given moment (i.e. the number of participants minus the number of processors that terminated) is always bounded by n.

3 The Allocation Problem

Consider a set R of $r \geq n$ items numbered 1 to r to allocate to the processors. In the allocation problem, we would like each processor $p \in P$ to output a set $D[p] \subseteq R$ such that $\{D[p] \mid p \in P\}$ is a partition of R. We require that if k processors participate, then the allocation forms a partition of the first $f(k)$ items, for some strictly increasing function $f : \mathbb{N} \to \mathbb{N}$ such that $f(0) = 0$ and $f(n) = r$ (hence there are at least as many items as participants, and all items are allocated if all processors participate). We also require that each processor get at least one item.

Definition 1. *In the allocation task, processors have no input and each processor p must output a set $D[p] \subseteq R$ such that: $D[p] \neq \emptyset$, and if a set Q of k processors participate then $\{D[p] \mid p \in Q\}$ must be a partition of $\{1, 2, \ldots, f(k)\}$.*

Given a solution to allocation, 2 processors can solve the consensus problem as follows. A processor p first posts its consensus proposal to shared-memory, and then participates in allocation and obtains an output $D[p]$. If $1 \in D[p]$ then p decides its own proposal. Otherwise, p decides the proposal of the only other processor q. Observe that when a processor p is the only participant, p must necessarily obtain $1 \in D[p]$ because, according to the definition of allocation, we must have $D[p] = \{i \mid 1 \leq i \leq f(1)\}$. Therefore, if a processor p sees $1 \notin D[p]$, then there must be another participant, and its proposal must be posted to shared-memory. In the case of a system of 2 processors, the other participant q is determined, and it must see $1 \in D_q$, because $\{D[p], D[q]\}$ must be a partition of $\{i \mid 1 \leq i \leq f(2)\}$, and therefore decide its own value.

Therefore both participants decide the same value. This shows that allocation has consensus power at least 2.

The allocation problem can be solved for any f using an array of n test-and-set objects $\{T[i] \mid 1 \leq i \leq n\}$. To solve allocation, a processor p accesses the test-and-set objects one-by-one, in order, and stops at the first test-and-set object $T[i]$ that it wins, returning the set of items $D[p] = \{j \mid f(i-1) < j \leq f(i)\}$. This algorithm is presented in the PlusCal language in Figure 1. Since allocation is solvable using test-and-set, which is implementable from 2-processors consensus, it has consensus power at most 2.

Theorem 1. *The allocation problem has consensus power 2.*

```
--algorithm Allocation{
    variables
        the outputs of the processors:
        D = [p ∈ P ↦ Bot] ;
        return value of TestAndSet procedure:
        ret = [p ∈ P ↦ Bot] ;
    process ( p ∈ P ) variables j = 1 ; {
l1:     while ( j ≤ N ) {
            call TestAndSet(j) ;
l2:         if ( ret[self] ) {
                if ( j = 1 ) D[self] := 1 .. f[1]
                else D[self] := (f[j − 1] + 1) .. f[j]
                goto "Done" }
            else j := j + 1 }  }  }
```

Figure 1: Algorithm for solving allocation.

4 Single-Use Assignment

Given a function $f : \mathbb{N} \to \mathbb{N}$, we formally define the assignment task as follows.

Definition 2. *In the assignment task, each processor must output a function $a[p] : D[p] \to P$ whose domain $D[p]$ is a set of items and such that:*

Fairness: *For every processor p, there is $r \in D[p]$ such that $a[p][r] = p$.*

Consistency: *For every processors p and q, if $r \in D[p]$ and $r \in D[q]$ then $a[p][r] = a[q][r]$.*

Non-Triviality: *If a set Q of k processors participate, then*

1. *for every $p \in Q$, $D[p] \subseteq \{1, \ldots, f(k)\}$ and $a[p]$ ranges over Q, and*
2. *for every item $i \in \{1, \ldots, f(k)\}$, there is a processor $p \in Q$ such that $i \in D[p]$.*

When a processor p terminates with output $a[p] : D[p] \to P$ we say that p announces the items in $D[p]$. This definition is a formalization of the intuitive definition given in the introduction, and restarted below.

Fairness: every processor p announces an assignment in which it gets at least one item.

Consistency: if two processors announce an assignment for item i, then they assign i to the same processor.

Non-Triviality: for every k, if k processors participate, then

1. only the first $f(k)$ items may be assigned, and
2. if all k processors terminate, then every item in $\{1, \ldots, f(k)\}$ is announced.

We now present an algorithm for solving the assignment task assuming that $f(k) = 2k - 1$ and $|R| = 2n - 1$. As noted in the introduction, this matches a lower bound obtained by reducing the renaming problem to the assignment problem. The algorithm uses immediate snapshot and adaptive-renaming sub-routines. A formalization of the algorithm in PlusCal appears in Figure 2.

A processor p first writes in shared-memory that it participates, and then takes an immediate snapshot (label *l1*). Then p invokes an instance of adaptive renaming in which only the members of p's immediate snapshot participate, obtaining the output $Name(p)$ (label *l2*). Processor p then considers the item number $2\big|is[p]\big| - Name(p)$ assigned to itself and writes it to shared memory in the variable $firstItem[p]$ (label

```
--algorithm $SingleUseAssignment${
    variables
        $participating = [p \in P \mapsto \text{FALSE}]$ ;
        $firstItem = [p \in P \mapsto Bot]$ ;   variable to post first item to shared memory.
        $is = [p \in P \mapsto Bot]$ ;   immediate snapshot output.
        $name = [i \in \text{SUBSET } P \mapsto [p \in P \mapsto Bot]]$ ;   renaming instances output.
        $a = [p \in P \mapsto Bot]$ ;   processor outputs
    define {
        $Name(p) \triangleq name[is[p]][p]$
        $Assign(Participant) \triangleq$
            LET $Posted \triangleq \{i \in Item : \exists p \in Participant : firstItem[p] = i\}$
                $Domain \triangleq 1 \,..\, 2 * Cardinality(Participant) - 1$
                $Free \triangleq Domain \setminus Posted$
                The free item $i$ has position k when it is the kth smallest free item:
                $Position(i) \triangleq Cardinality(\{j \in Free : j \leq i\})$
                A processor has rank $i$ when its first item is the ith smallest posted item:
                $Rank(p) \triangleq Cardinality(\{q \in Participant : firstItem[q] \leq firstItem[p]\})$
            IN  $[i \in Domain \mapsto$ IF $i \in Posted$
                    THEN CHOOSE $p \in Participant : firstItem[p] = i$
                    ELSE CHOOSE $p \in Participant : Rank(p) = Position(i)]$ }
fair process ( $proc \in P$ ) {
$l1$:   $participating[self] := \text{TRUE}$ ;
        call $ImmediateSnapshot()$ ;
$l2$:   call $Renaming(is[self])$ ;
$l3$:   $firstItem[self] := 2 * Cardinality(is[self]) - Name(self)$ ;
$l4$:   with ( $Participant = \{p \in P : participating[p]\}$ )
        if ( $\exists p \in Participant : firstItem[p] = Bot$ ) $a[self] := [i \in \{firstItem[self]\} \mapsto self]$
        else $a[self] := Assign(Participant)$ ; } }
```

Figure 2: Read-write algorithm solving single-use assignment.

l3). At this point we say that p *posted* its first item; moreover, if $firstItem[q] = i$ for some $q \in P$ and $i \in R$, then we say that i has been posted. Finally, at label *l4*, p checks whether there is a participant that did not post its first item. If this is the case, then p announces only its first assigned item, i.e. it outputs $a[p] = [firstItem[p] \mapsto p]$.

Otherwise, when all the k participants posted their first item, p announces the assignment of all first $2k-1$ items as follows. Let us say that an item among the first $2k-1$ is free if it has not been posted. Processor p assigns every posted item i to the processor q that posted $firstItem[q] = i$ (this processor is unique by Lemma 1 below), and p assigns the ith free item to the processor q that posted the ith biggest item (also unique by Lemma 1).

To show that Figure 2 solves the assignment task, we need the following definitions. Consider a run of the algorithm in which a set Q of k processors participate, and consider the immediate snapshots $IS_1, \ldots, IS_m$ obtained by the participants, ordered by inclusion, and let $IS_0 = \emptyset$. Define the sequence of sets of processors $G_1, \ldots, G_m$ where $G_i = IS_i \setminus IS_{i-1}$, and let $G_0 = \emptyset$. Finally, define the sequence of intervals $I_1, \ldots, I_m$ where $I_i = \{2|IS_{i-1}| + 1, \ldots, 2|IS_i| - 1\}$. Note that if $i < j$ then $Max\,(I_i) \leq Min\,(I_j)$, and that $|I_i| = 2|G_i| - 1$. Those definitions are best understood by considering the following lemmas.

Lemma 1. *For every $i \in \{1, \ldots, m\}$, the members of G_i obtain unique first items in the interval I_i, and only processors in G_i obtain items in I_i.*

Proof. By definition of immediate snapshot, a processor obtains the immediate snapshot IS_i if and only if it belongs to G_i. Moreover, by definition of the algorithm, only the members of G_i ever access the adaptive-renaming instance for the set of processors I_i. Therefore, by property of adaptive renaming, the members of G_i obtain unique names in $\{1, \ldots, 2|G_i| - 1\}$, and only processors in G_i obtain items in $I_i = \{2|I_{i-1}| + 1, \ldots, 2|I_i| - 1\}$. Thus, by definition of the algorithm at label *l3*, the members of G_i obtain unique first items in I_i, and only processors in G_i obtain items in I_i. $\square$

Note that Lemma 1 implies that every processor gets a unique first item. Let $\mathit{PostedBy}(i) = p$ if i is posted by processor p in the run under consideration and $\mathit{PostedBy}(i) = \bot \notin P$ otherwise. By Lemma 1, $\mathit{PostedBy}(i)$ is well-defined.

Lemma 2. *Every processor p that takes the else branch at label l4 does so with Participant $= IS_i$ for some $i \in \{1, \ldots, m\}$.*

Proof. By definition of the algorithm, when p takes the else branch at label *l4*, every processor invoked and returned from immediate snapshot at label *l1*. Therefore, by definition of immediate snapshot, at least one processor obtained an immediate snapshot containing all the participants. $\square$

Lemma 3. *If p announces i and $\mathit{PostedBy}(i) = q$, then $a[p][i] = q$.*

Proof. First, since $\mathit{PostedBy}(i) = q$ when p announces its output, note that q posted its item before p reached *l4*. Consider two cases. First, suppose $p = q$. Therefore, if p takes the if branch at *l4*, then it announces $[i \mapsto p]$ and we are done. If p takes the else branch, then by definition of the *Assign* operator, we have $a[p][p] = i$, and we are done.

Second, suppose that $p \neq q$. If p takes the if branch at *l4*, then it announces $[j \mapsto p]$ where $\mathit{PostedBy}(j) = p$. By Lemma 1, we must have $i \neq j$, and we are done. If p takes the else branch at *l4*, then by Lemma 2 there is $j \in \{1, \ldots, m\}$ such that $a[p] = \mathit{Assign}(IS_j)$, IS_j is exactly the set of participants at this point, and all members of IS_j posted their first item. Therefore either (a) q did not participate yet and $q \notin IS_j$, or (b) $q \in IS_j$ and q posted its first item.

In case (a), $q \in G_l$ for $l > j$; therefore, by Lemma 1, item i is strictly greater than $\mathit{Max}(I_i)$. Moreover, the domain of $a[p] = \mathit{Assign}(IS_j)$ is I_j by definition of the *Assign* operator. Thus $i \notin a[p]$ and p does not announce i, a contradiction.

In case (b) we have $a[p][q] = i$, by definition of the *Assign* operator, and we are done. $\square$

Lemma 4. *If p takes the else branch at l4 before q takes the same else branch, then for every item i announced by p, $i \in D[q]$ and $a[q][i] = a[p][i]$.*

Proof. By Lemma 2, there are $j < k \in \{1, \ldots, m\}$ such that all members of IS_k posted their first item by the time q takes the else branch at *l4*, and $a[p] = \mathit{Assign}(IS_j$ and $a[q] = \mathit{Assign}(IS_k)$.

Note that at the time p takes the else branch at *l4*, all members of IS_j posted their first item. Therefore, by definition of the *Assign* operator, the rank of a processors $p \in IS_j$ is the same in the definition of $\mathit{Assign}(IS_j)$ and in the definition $\mathit{Assign}(IS_k)$. Moreover, by Lemma 1, the free items in the range $\{1, \ldots, 2|I_j| - 1\}$ do not change after p takes the else branch at *l4*. Thus, by definition of the *Assign* operator, if i is announced by p then $a[q][i] = a[p][i]$. $\square$

Lemma 5. *At least one participant p_l finds at label l4 that all participants posted their first item.*

Proof. The last participant to post its first item finds at *l4* that all participants posted their first item. $\square$

Theorem 2. *The single-use assignment algorithm of Figure 2 solves the assignment task using only registers.*

Proof. The algorithm clearly uses only registers, and so does its immediate-snapshot and adaptive-renaming sub-routines. Moreover, its immediate snapshot and its adaptive-renaming sub-routines are wait-free, and every processor performs at most 4 atomic steps in the algorithm, therefore the algorithm is wait-free. It remains to show that outputs satisfy the assignment task.

Notice that every processor announces at least the item that it posted. Therefore, the Fairness property of the assignment task is satisfied.

To show the Consistency property, consider two processors p and q that both announce item i. If both p and q take the if branch at label *l4*, then by Lemma 1 they cannot both announce i. Therefore, without loss of generality, either p takes the if branch and q takes the else branch, or both take the else branch at label *l4*. Suppose p takes the if branch and q takes the else branch. Then, i must be the item posted by p, and by Lemma 3 both p and q announce the same assignment for i, and we are done. Suppose both p and q take the else branch at label *l4*, and, without loss of generality, that p does so before q. Then, by Lemma 4, if i is in the domain of $a[p]$, then $a[q][i] = a[p][i]$, and we are done.

Part (a) of the Non-Triviality property follows from Lemma 1 because for every $i \in \{1, \ldots, m\}$, $I_i \subseteq \{1, \ldots, 2k-1\}$. Part (b) follows from Lemma 5: the last participant to post its item, p_l, sees all k participants and takes the else branch at *l4*; therefore, by the definition of the *Assign* operator, p_l announces all the first $2k-1$ items. $\square$

Note that the algorithm is as fair as can be: it guarantees that a processor gets at least one item and at most 2. Since $2k-1$ items are assigned when k processors participate, this is optimal.

Finally, note that we can modify the algorithm to work with any function f such that $f(1) \geq 1$ and $f(k) - f(j) \geq 2(k-j) - 1$ for every $k > j \in \mathbb{N}$. For this, we first change the first item of a processor p to be the item number $f(|IS(p)|) - Name(p)$ (at label *l3*), and we change how a processor p, that sees all the renaming output of the participants (at label *l4*, else branch), allocates the remaining items. Let k be the number of participants that p sees when it takes its step at label *l4*. Processor p uses a larger domain $\{i \mid 1 \leq i \leq f(k)\}$, and, for every j from 1 to k, p allocates the next $f(j+1) - f(j) - 1$ free items to the processor of rank j (when $j = k$, there may not remain enough items, and in this case only the remaining free items are assigned).

5 Long-Lived Assignment

We now consider solving the assignment task for infinitely many processors, assuming that the number of concurrently active processors is bounded by a constant. As explained in Section 2, when the set of processors is fixed, this allows processors to repeatedly invoke the task by picking a fresh identifier for each new invocation.

The definition of the task is the same as in the single-use case, except that the set of processors P is infinite and the set of items R is also infinite. Items and processors are numbered $1, 2, \ldots$, and we assume that $f : \mathbb{N} \to \mathbb{N}$ is such that $f(k) = 2k - 1$.

Note that we do not consider releasing items already assigned, but only assigning new items from an infinite stream of items. The non-triviality condition of the assignment task ensures that only the first $2k-1$ items may be assigned when k processors participate. In the long-lived setting, this means that processors cannot get items arbitrarily far in the stream: if m is the number of times that processors invoked the task, then only the first $2m-1$ items may be assigned.

Obtaining a long-lived assignment algorithm is simple: it suffices to replace the immediate-snapshot subroutine in the algorithm of Section 4 by an immediate snapshot for infinitely many processors (but bounded concurrency), as provided, e.g., by Afek et al. [1].

Note that, in the long-lived setting, if processors progress at the same speed then the stream of items will be consumed without leaving holes. If not, some items may be left unassigned while more and more items farther in the stream are assigned. In fact, a processor can arbitrarily delay the allocation of an arbitrary large number of items: if a processor p, after reaching label *l2*, delays its posting of its first item, then after p reached *l2*, every processor will only ever get a single item because, at label *l4*, every processor will always find that p did not write *firstItem*$[p]$.

We now present an optimization of the long-lived algorithm in which a processor that stops can prevent the allocation of at most n items, where n is the fixed number of processors that repeatedly invoke the algorithm. To achieve this property, we first introduce a new label *l1b*, immediately after *l1*, where a processor posts to shared memory the immediate snapshot it

```
--algorithm $LongLivedAssignment${
    variables
        $name = [i \in \text{SUBSET } P \mapsto [p \in P \mapsto Bot]]$ ;   return values from renaming.
        $is = [p \in P \mapsto Bot]$ ;   return values from immediate snapshot.
        $postedIS = [p \in P \mapsto Bot]$ ;
        $firstItem = [p \in P \mapsto Bot]$ ;
        $a = [p \in P \mapsto Bot]$ ;   processor outputs
    define {
        $Name(p) \triangleq name[is[p]][p]$
        A frame for $p$ consists of two immediate snapshots $IS1$ and $IS2$ such that $p$ is in $IS2$ but
        not in $IS1$
        $Frame(p, PostedIS) \triangleq \text{LET } IS \triangleq PostedIS \cup \{\{\}\} \text{IN}$
            $\{\langle IS1, IS2\rangle \in IS \times IS : p \in IS2 \land p \notin IS1\}$
        $MaxFrame(F) \triangleq \text{CHOOSE } \langle IS1, IS2\rangle \in F : \forall I \in F : IS1 \subseteq I[1] \land I[2] \subseteq IS2$
        Computing the assignment on an frame $\langle Low, High\rangle$:
        $Assign(Low, High) \triangleq$
            $\text{LET } ItemsPosted \triangleq \{i \in Item : \exists p \in High \setminus Low : firstItem[p] = i\}$
                $Domain \triangleq 2 * Cardinality(Low) + 1 \,..\, 2 * Cardinality(High) - 1$
                $Free \triangleq Domain \setminus ItemsPosted$
                $Position(i) \triangleq Cardinality(\{j \in Free : j \leq i\})$
                $Rank(p) \triangleq Cardinality(\{q \in High \setminus Low : firstItem[q] \leq firstItem[p]\})$
            $\text{IN}$   $[i \in Domain \mapsto \text{IF } i \in ItemsPosted$
                        $\text{THEN CHOOSE } p \in High \setminus Low : firstItem[p] = i$
                        $\text{ELSE CHOOSE } p \in High \setminus Low : Rank(p) = Position(i)]$
    }
fair process ( $proc \in P$ ) {
l1:     call $ImmediateSnapshot()$ ;
l1b:    $postedIS[self] := is[self]$ ;
l2:     call $Renaming(is[self])$ ;
l3:     $firstItem[self] := 2 * Cardinality(is[self]) - Name(self)$ ;
l4:     with ( $PostedIS = \{postedIS[p] : p \in P\} \setminus \{Bot\}$ )
        if ( $\forall F \in Frame(self, PostedIS) : \exists p \in F[2] \setminus F[1] : firstItem[p] = Bot$ )
            $a[self] := [i \in \{firstItem[self]\} \mapsto self]$
        else with ( $Complete = \{F \in Frame(self, PostedIS) : \forall p \in F[2] \setminus F[1] : firstItem[p] \neq Bot\}$,
                    $MaxF = MaxFrame(Complete)$ )
            $a[self] := Assign(MaxF[1], MaxF[2])$
    }
 }
```

Figure 3: Read-write algorithm solving assignment for infinitely many processors when at most n are concurrent. The number of items blocked by a slow processor is bounded by n.

obtained at label *l1*. Second, we modify the code at label *l4* as follows: a processor p at label *l4* first checks whether it can find two immediate snapshots IS_1 and IS_2 that have been posted to shared memory such that (1) $p \in IS_2$, (2) $p \notin IS_1$, and (3) all members of $IS_2 \setminus IS_1$ have written their first item to shared memory. Two immediate snapshots $\langle IS_1, IS_2\rangle$ with those properties are called a complete frame for p. If p cannot find such a complete frame $\langle IS_1, IS_2\rangle$, then it terminates with the output $a[p] = [firstItem[p] \mapsto p]$. Otherwise, p picks $\langle IS_1, IS_2\rangle$ satisfying conditions (1), (2), and (3) and such that $IS_2 \setminus IS_1$ is maximal among the complete frames for p. Finally, p uses the same ranking mechanism as in the single-use case to compute its output, using the $Assign(IS_1, IS_2)$ operator, except that it restricts the domain of its output to the items in the range $\{2|IS_1| + 1 \ldots 2|IS_2| - 1\}$. A PlusCal formalization of the optimized long-lived algorithm appears in Figure 3. The algorithm correctness relies on essentially the same arguments as Theorem 2:

Theorem 3. *The long-lived assignment algorithm of Figure 3 solves the long-lived assignment task using only registers.*

Theorem 4. *When at most n processors can be active at the same time, the long-lived assignment algorithm of Figure 3 ensures that a processor that stops prevents the allocation of at most n items.*

Proof. A processor can block the assignment of some items only if it stops after its first step but before it posts its first item. Consider a run in which p does so. Consider the immediate snapshots $IS_1, \ldots, IS_m$ obtained by the participants, ordered by inclusion. Let IS_i, $i \in \{1, \ldots, m\}$, be the biggest immediate snapshot in the run such that $p \notin IS_i$. By the definition of the algorithm at label *l4*, processor p prevents the allocation of all items in $I_{i+1} = \{2|IS_i| + 1, \ldots, 2|IS_{i+1}| - 1\}$ that are not posted by any processor. By property of immediate snapshot, at worse, $IS_{i+1} - IS_i = n$, thus $|I_{i+1}| \leq 2n - 1$. Among those, $n-1$ will be posted if only p stops, and therefore p can prevent the allocation of n items at most. $\square$

6 Conclusion and Related Work

We have shown that allocating items to asynchronous processors requires primitives of consensus power 2, but that a new variation on the allocation problem, the assignment problem, is solvable read-write wait-free. Moreover, we have presented a long-lived assignment algorithm in which a failed processor can only prevent the allocation of a constant number of items. Long-lived assignment can readily be solved using a mutual exclusion algorithm such as Lamport's Bakery algorithm [17]. However, in the Bakery algorithm, a slow processor can arbitrarily delay the whole system even when the slow processor is not in its critical section. In contrast, the optimized long-lived assignment algorithm presented in Section 5 ensures that a slow processors delays the assignment of at most n items while the other processors suffer no delay. The long-lived assignment problem and its solution may therefore be of interest to implement resource allocators, work queues, and stream processing systems.

Below we briefly survey related work on resource allocation and on the adaptive-renaming problem, whose solution we rely on in solving read-write assignment.

In the dining-philosophers problem [9] or the mutual-exclusion problem, a number of resources are to be acquired to perform a task and then released, but there are not enough resources for all processor to perform their task at the same time. In the dining philosophers problem, processors are placed in a ring with one resource between each neighboring pair of processors; a processor contends for the two resources immediately adjacent to it in the ring. In the mutual-exclusion problem, processors share a single resource that they all contend for. Under the assumption that all processors progress fairly, solutions to those problems must guarantee that no processor starve. The specification of the problem does not leave room for failures, as it becomes trivially unsolvable when some resources are not released. Failures complicate allocation and increase the number of resources necessary to make the problem solvable. At the very least, enough resources should be available to satisfy one processor should all others fail when holding resources.

The L-exclusion [11] problem is a generalization of mutual exclusion in which at most $L < n$ processors can be the critical section simultaneously. Here, the failure of L processors in the critical section trivially halts the system, but the algorithm presented in [11] may also deadlocks if L processors fail when trying to enter the critical section.

In the m-renaming problem [5], n processors must exclusively acquire a name between 1 and m, under one of two non-triviality conditions: either the initial identifiers of the processors are assumed to come from an unbounded namespace, or, in adaptive renaming, the range of names used must depend on the set of processors that participate. The adaptive renaming problem is not solvable read-write wait-free when k processors out of n must perform renaming using the first $2k - \lceil k/(n-1) \rceil$ names [12]. With $2k-1$ names, several wait-free adaptive renaming algorithms are known [5, 6], as well as long-lived wait-free adaptive renaming algorithms in which names can be released by their owner [21, 22]. The musical chairs problem [3] is a variant of renaming in which processors come with preferences and must rename themselves such that a processor gets his preferred name if no other participant has the same preference. The musical chairs problem is equivalent to renaming. The assignment problem defined in this paper differs from renaming or musical chairs in that all items must be assigned to some processor; long-lived assignment differs from the long-lived version of renaming in that, instead of releasing items, processors consume items from an infinite stream.

Castaneda et al. [8] study single-use assignment under preferences and constraints. The problem they study is a generalization of the renaming and musical chairs [3] problems, where processors must choose names subject to preferences that must be satisfied in the absence of conflict, and subject to constraints that precludes certain assignments. As in the renaming problem, the problem they formulate has no requirements to assign all the items. While the algorithms presented in this paper rely on renaming, Castaneda et al. exhibit instances of the coordination task under preferences and constraints in which renaming-based solutions may not be optimal.

L-assignment (or "distinct CS") [7, 4] is a variant of L-exclusion in which processors entering the critical section must additionally be assigned a unique slot out of a number L of slots. The At-Most-Once problem of Kentros et al. [16] is closely related to L-assignment and renaming. It is a single-use allocation problem in which some items may be left unallocated. Kentros et al. give bounds on the number of items that can be allocated read-write out of the total number of items (called the efficiency of an algorithm), depending on the number of processor failures. In contrast to our work, Kentros et al. do not consider the possibility for a processor to leave with a partial allocation whose missing items will be revealed later by other processors. In follow-up work, Kentros et al. study deterministic solutions to the At-Most-Once problem that minimize the work that processors have to perform [15], as well as randomized solutions under fair scheduling assumptions [14].

The Write-All problem [13], introduced by Kanellakis and Schwarzmann, all positions in a shared array must be set using the minimal amount of work (as measured in number of steps). The Write-All problem differs from the allocation problem in that some positions in the array may be set by multiple processors, while an item in the allocation and assignment problems must be allocated exclusively to one processor. In the terminology of this paper, the Write-All problem requires each item to be allocated at least once, while the allocation and assignment problems requires each item to be allocated at most once. Dwork et al. study the Do-All problem [10], a variant of the Write-All problem in message-passing systems.

The assignment algorithms presented in this paper are inspired by the adaptive-renaming algorithm of Borowsky and Gafni [6]. In this recursive algorithm, processors use immediate snapshot to split themselves in disjoint groups that each is implicitly assigned a unique part of the namespace; then, each group recursively solves adaptive-renaming among the members of the group. The part of the namespace allocated to a group is sufficiently big to place the outputs of the corresponding adaptive-renaming sub-problem in that part of the namespace. We reuse the idea of using immediate snapshot to split processors into groups that are implicitly assigned a unique part of the namespace that is big enough to solve renaming among the group member.

7 Acknowledgments

Carole Delporte and Hugues Fauconnier were supported by the Agence Nationale de la Recherche, project DESCARTES, under grant agreement ANR-16-CE40-0023. This research was partially supported by Len Blavatnik and the Blavatnik Family foundation. This material is based upon work supported by the National Science Foundation under Grant No.

1655166.

References

[1] Yehuda Afek and Eytan Weisberger. “The instancy of snapshots and commuting objects”. In: *Journal of Algorithms* 30.1 (1999), pp. 68–105. (Visited on 01/22/2017).

[2] Yehuda Afek et al. “Atomic Snapshots of Shared Memory”. In: *Journal of the ACM* 40.4 (Sept. 1993), pp. 873–890. ISSN: 0004-5411. (Visited on 11/16/2016).

[3] Y. Afek et al. “Musical Chairs”. In: *SIAM Journal on Discrete Mathematics* 28.3 (Jan. 2014), pp. 1578–1600. ISSN: 0895-4801.

[4] Hagit Attiya et al. “Renaming in an asynchronous environment”. In: *Journal of the ACM (JACM)* 37.3 (1990), pp. 524–548. (Visited on 02/03/2017).

[5] H. Attiya et al. “Achievable cases in an asynchronous environment”. In: *28th Annual Symposium on Foundations of Computer Science, 1987.* Oct. 1987, pp. 337–346.

[6] Elizabeth Borowsky and Eli Gafni. “Immediate Atomic Snapshots and Fast Renaming”. In: *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing.* PODC ’93. ACM, 1993, pp. 41–51. (Visited on 11/16/2016).

[7] J. E. Burns and G. L. Peterson. “The Ambiguity of Choosing”. In: *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing.* PODC ’89. New York, NY, USA: ACM, 1989, pp. 145–157.

[8] Armando Castañeda et al. “Asynchronous Coordination Under Preferences and Constraints”. en. In: *Structural Information and Communication Complexity.* Vol. 9988. Lecture Notes in Computer Science. Springer, Cham, July 2016, pp. 111–126. (Visited on 07/18/2017).

[9] E. W. Dijkstra. *Two starvation free solutions of a general exclusion problem, 1978.* EWD, 1977.

[10] Cynthia Dwork, Joseph Y. Halpern, and Orli Waarts. “Performing work efficiently in the presence of faults”. In: *SIAM Journal on Computing* 27.5 (1998), pp. 1457–1491.

[11] Michael J. Fischer et al. “Distributed FIFO Allocation of Identical Resources Using Small Shared Space”. In: *ACM Trans. Program. Lang. Syst.* 11.1 (Jan. 1989), pp. 90–114. ISSN: 0164-0925.

[12] Eli Gafni, Michel Raynal, and Corentin Travers. “Test & set, adaptive renaming and set agreement: a guided visit to asynchronous computability”. In: *26th IEEE International Symposium on Reliable Distributed Systems, 2007.* IEEE, 2007, pp. 93–102. (Visited on 01/21/2017).

[13] Paris C. Kanellakis and Alex A. Schwarzmann. “Efficient parallel algorithms can be made robust”. In: *Distributed Computing* 5.4 (1992), pp. 201–217.

[14] Sotirios Kentros, Chadi Kari, and Aggelos Kiayias. “The strong at-most-once problem”. In: *International Symposium on Distributed Computing.* Springer, 2012, pp. 386–400.

[15] Sotirios Kentros and Aggelos Kiayias. “Solving the at-most-once problem with nearly optimal effectiveness”. In: *International Conference on Distributed Computing and Networking.* Springer, 2012, pp. 122–137.

[16] Sotirios Kentros et al. “At-most-once semantics in asynchronous shared memory”. In: *International Symposium on Distributed Computing.* Springer, 2009, pp. 258–273.

[17] Leslie Lamport. “A new solution of Dijkstra’s concurrent programming problem”. In: *Communications of the ACM* 17.8 (1974), pp. 453–455.

[18] Leslie Lamport. “On interprocess communication: Part II”. en. In: *Distributed Computing* 1.2 (June 1986), pp. 86–101. ISSN: 0178-2770, 1432-0452. (Visited on 11/16/2016).

[19] Leslie Lamport. "The PlusCal Algorithm Language." In: *ICTAC*. Vol. 5684. Springer, 2009, pp. 36–60.

[20] Michael C. Loui and Hosame H. Abu-Amara. "Memory requirements for agreement among unreliable asynchronous processes". In: *Advances in Computing research* 4.163-183 (1987), p. 31.

[21] Mark Moir and James H. Anderson. "Wait-free algorithms for fast, long-lived renaming". In: *Science of Computer Programming* 25.1 (Oct. 1995), pp. 1–39. ISSN: 0167-6423.

[22] Mark Moir and Juan A. Garay. "Fast, long-lived renaming improved and simplified". en. In: *Distributed Algorithms*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Oct. 1996, pp. 287–303. (Visited on 07/22/2017).

[23] Gary L. Peterson. "Myths about the mutual exclusion problem". In: *Information Processing Letters* 12.3 (1981), pp. 115–116.

[24] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model checking TLA+ specifications". In: *CHARME*. Vol. 99. Springer, 1999, pp. 54–66.