Allocation adaptative de registres en utilisant un nombre linéaire de registres

Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, Leslie Lamport

Allocation adaptative de registres en utilisant un nombre linéaire de registres †

Carole Delporte-Gallet[1] ‡ and Hugues Fauconnier[1] § and Eli Gafni[3] and Leslie Lamport[4]

[1] *LIAFA, U. Paris Diderot, France*
[2] *Computer Science Department, UCLA, USA*
[3] *Microsoft Research*

On présente un algorithme adaptatif dans lequel les processus utilisent des registres multi écrivains multi lecteurs. Cet algorithme permet à chaque processus d'obtenir un accès exclusif à un registre dont il sera l'écrivain unique et que tous les processus pourront lire. L'algorithme est adaptatif : il ne connait pas a priori le nombre de processus qui vont demander un accès exclusif en écriture à un registre. C'est le premier algorithme permettant d'obtenir ce résultat en utilisant des registres dont le nombre est une fonction linéaire du nombre de participants. Les précédents algorithmes adaptatifs utilisent au moins $\Theta(n^{3/2})$ registres.

Keywords: shared memory, read/write registers, distributed algorithms, wait-free, space complexity, renaming.

1 Introduction

One way to implement multiprocess synchronization is by providing each process with a single-writer, multi-reader atomic register (SWMR) that it can write and other processes can read. We present an adaptive algorithm to implement such a system of registers with an array of multi-writer multi-reader atomic (MWMR) registers whose length is linear in the number of participating processes. The algorithm is non-blocking unless an unbounded number of processes initiate operations.

An adaptive algorithm, also called a uniform algorithm [Gaf02], is one that does not know the number of potentially participating processes. Equivalently, it is an algorithm whose cost is a function not of the total number of processes but of the number of processes that actually participate in the algorithm. For the SWMR registers, this is the number of processes that actually perform a read or write operation. Our goal is to minimize the number of MWMR registers, and our algorithm uses a number that is linear in the number of participants. No *a priori* bound on this number is assumed.

Why do we find this algorithm interesting ? There are simpler algorithms that assume stronger communication primitives—for example, test and set registers—but MWMR registers are the weakest ones for which we know that an adaptive algorithm is possible. More efficient randomized algorithms are possible, but our algorithm is always correct, not just correct with high probability. There is a trivial way to implement a collection of SWMR registers with an array C of MWMR registers. The i^{th} process simply uses $C[i]$ as its register. Of course, this algorithm uses an unbounded number of registers. The obvious way to make the number of registers linear in the number of participating processes is by having the processes first execute an adaptive renaming algorithm [ABND$^+$90, BG93] in which each participating process is assigned a unique number from 0 to M for some M that depends linearly on the number of participants. A process assigned the number j then uses $C[j]$ as its register. However, we know of few renaming algorithms that do not assume a collection of SWMR registers already allocated to processes [Asp10, AF02, MA95]. Those algorithms are all based on the grid-network of "splitters" proposed by Anderson and Moir [MA95]. Of these,

† This work has been accepted for publication at DISC2013 [DGFGL13].
‡ Supported by ANR DISPLEXITY.
§ Supported by ANR DISPLEXITY.

```
--algorithm GFX
{ variables A1 = [i ∈ Nat ↦ {}], result = [p ∈ Proc ↦ {}] ;
  process (Pr ∈ Proc)
    variables known = {self}, notKnown = {} ;
    { a : known := known ∪ NUnion(A1) ;
          notKnown := {i ∈ 0 .. (Cardinality(known)) : known ≠ A1[i]} ;
          if (notKnown ≠ {})
            { b : with (i ∈ notKnown) {A1[i] := known} ;
                  goto a
            }
          else {result[self] := known} ;
    }
}
```

FIGURE 1: Algorithm *GFX*.

the more space-efficient is an improvement of Aspnes [Asp10] that requires $\Theta(k^{3/2})$ MWMR registers for k participating processes. Even though the renaming algorithm is used only to determine the assignment of processes to elements of the array C, the values in those $\Theta(k^{3/2})$ registers must be maintained forever because additional processes may enter the system at any time. (Reclaiming the space requires knowing an *a priori* bound on the number of processes that might participate.) Thus, our algorithm is the first that implements a collection of SWMR registers with $O(k)$ MWMR registers.

Almost all previous methods for making an algorithm adaptive start by using one of several renaming algorithms [AAD$^+$93, AST99, And94, ABND$^+$90, BG93]. It has generally been assumed that this is the only way to implement an adaptive algorithm [AW98]. Based on an idea in [DGFGR13], our implementation avoids the use of a renaming algorithm to begin reliable communication. Instead, participating processes first announce their presence by using a non-blocking one-shot limited-snapshot algorithm that we call the GFX (Generalized Fast eXclusion) protocol, which can be viewed as generalizing [Lam87] from 1-concurrency to k-concurrency. The snapshot is limited to having the property that two snapshots of the same size coincide. It need not ensure that snapshots of different sizes are related by containment. To perform a read or write operation to a register, a process first reads the posted snapshots to find the number n of participants that have announced their presence, and it executes an algorithm [DGFGR13] that assumes at most n processes. It then reads the number of participants again, finishing the operation if that number still equals n. Otherwise, the process repeats the n-process algorithm for the new value of n. While we use this approach to implement renaming, it can be used to provide an adaptive implementation of any task.

By using our adaptive algorithm for implementing a collection of SWMR registers, we can solve any task under the assumption of finite arrival [GMT01]. In particular, using existing algorithms, we can implement adaptive renaming with a linear range [ABND$^+$90, BG93]. This in turn allows us to allocate unique registers to processes with a number of registers linear in the number of participants. With register allocation, we can implement a collection of SWMR registers with wait-free read and write operations rather than just non-blocking ones. For many tasks of high read-write complexity, doing renaming first may reduce the step complexity of an adaptive algorithm.

We ignore time complexity—the number of steps taken by the algorithm. Our algorithm is executed just once, to assign SWMR registers to processes ; it adds nothing to the cost of using those registers. Since space used by an adaptive algorithm cannot be reclaimed, it is perhaps more important than time complexity. Still, optimal time complexity is an interesting problem that remains unsolved.

In the non-adaptive case, it has been shown that at least n registers are required to implement n SWMR registers [DGFGR13], so the linear number of registers used by our algorithms is optimal up to a constant factor. We originally believed that adaptive algorithms required more than a linear number of registers, and we tried to derive such a lower bound on the number of registers, independent of their size. When the difficulty is caused by processes stepping on each other because of the lack of *a priori* coordination, size of the registers is not a factor. (See the lower bound for consensus [FHS98].) We were therefore surprised to discover our algorithm.

We precisely describe our algorithms in the PlusCal algorithm language [Lam09]. A PlusCal expression can be any TLA$^+$ formula [Lam02], and a PlusCal algorithm is automatically translated to a TLA$^+$ specification that defines the algorithm's formal meaning.

We have written formal, mechanically-checked TLA$^+$ correctness proofs of the safety properties of the

GFX and *SnapShot* algorithms. The complete proofs are available on the Web [Lam].

2 Algorithms

```
--algorithm SnapShot
{ variables result = [p ∈ Proc ↦ {}],
            A2 = [i ∈ Nat ↦ {}],  A3 = [i ∈ Nat ↦ {}] ;
  process (Pr ∈ Proc)
    variables myVals = {},  known = {},  notKnown = {},
              lnbpart = 0,  nbpart = 0,  nextout = {},  out = {} ;
    { a : with (P ∈ {Q ∈ SUBSET Proc :                          /*Specification of Algorithm GFX*/
                        ∧ self ∈ Q
                        ∧ ∀p ∈ Proc \ {self} :
                            ∨ Cardinality(result[p]) ≠ Cardinality(Q)
                            ∨ Q = result[p]
                } )
          { result[self] := P } ;
        A2[Cardinality(result[self]) − 1] := result[self] ;
      b : while ( TRUE )
          {   with (v ∈ Val) { myVals := myVals ∪ {v} } ;          /* snap(v) */
              known := myVals ∪ known ;
              nbpart := Cardinality(NUnion(A2)) ;
          c : lnbpart := nbpart ;
              known := known ∪ NUnion(A3) ;
              notKnown := {i ∈ 0 .. (nbpart − 1) : known ≠ A3[i]} ;
              if (notKnown ≠ {}) { d : with (i ∈ notKnown)
                                         { A3[i] := known } ;
                                       goto c }
              e :                  nbpart := Cardinality(NUnion(A2)) ;
                                   if (lnbpart = nbpart) {out := known}      /*returned value */
                                   else {goto c}
          }
    }
}
```

FIGURE 2: Algorithm *SnapShot.*

A sequence of SWMR registers is implemented using an algorithm we call $SnapShot$. This algorithm begins with Algorithm *GFX* that we describe below.

Algorithm *GFX*

Algorithm GFX, described in Figure 1, solves the following weaker version of the snapshot task [AAD$^+$93] : A process p that executes the algorithm must return a set F_p of participants such that

— $p \in F_p$ for any p.

— $|F_p| = |F_q|$ implies $F_p = F_q$ for any p and q, where $|F|$ is the cardinality of the set F.

The variables $known$ and $notKnown$ are local to $self$ (the current process) and cannot be read or written by other processes. Variable $known$ stores the set of processes known to process $self$, and $unKnown$ stores a set of array indices (natural numbers). The values of these process-local variables are arrays indexed by the set $Proc$. The other new notations used in this algorithm are : Nat is the set of natural numbers, $i \,..\, j$ is the set of integers k with $i \leq k \leq j$, the statement **with** $(x \in S)\{\Sigma\}$ executes Σ with an arbitrary element of S substituted for x and the operator $NUnion$ is defined by $NUnion(A) \triangleq \text{UNION}\{A[i] : i \in Nat\}$. Evaluation of that expression is implemented by atomically reading the array A. Observe that although $result$ is a global variable, $result[p]$ is accessed only by process p.

Algorithm *Snapshot*

The $SnapShot$ algorithm maintains a set S of values that is initially empty. It provides a $snap$ operation whose argument is a value v. Executing $snap(v)$ atomically adds v to S and returns the current value of S. This allow to simulate for each process a SWMR register.

Let's suppose that there is a *count* operation that a process p can call to learn the number of participants that can be executing a *snap* operation. To perform a *snap* operation, a process p first executes *count* to

obtain a bound n on the number of participants. It then writes in the first n registers of $A3$. If a read of $A3$ obtains a value F such that $A3[0] = \cdots = A3[n-1] = F$, process p executes the *count* operation again. If that execution returns the same number n of participants, then the *snap* operation completes and returns the value F. Otherwise, the process continues the procedure, replacing n with the new value returned by *count*.

We still have to implement the *count* operation. We do that by using algorithm GFX and a second array $A2$ of registers. When a participant p arrives, before performing any *snap* operation it (i) executes GFX to obtain a set S of participants, which includes itself, and (ii) writes (the processes in) S in $A2[|S|-1]$. The correctness property of GFX implies that no other value can ever be written in $A2[|S|-1]$. Since the processes written in $A2$ are all participants and every participant is written in $A2$, the set of all processes in $A2$ includes all participants that can write to $A3$. The *count* operation is then performed by reading $A2$ and counting the number of (distinct) processes read.

Algorithm $SnapShot$ appears in Figure 2. We have represented the code of GFX in $SnapShot$ by the corresponding code of its specification in TLA$^+$.

Références

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

[ABND+90] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3) :524–548, 1990.

[AF02] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2) :642–664, February 2002.

[And94] James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4) :175–195, 1994.

[Asp10] James Aspnes. Slightly smaller splitter networks. *CoRR*, abs/1011.3170, 2010.

[AST99] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived adaptive collect with applications. In *FOCS*, pages 262–272. IEEE Computer Society, 1999.

[AW98] Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.

[BG93] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *PODC*, pages 41–51. ACM Press, 1993.

[DGFGL13] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Leslie Lamport. Adaptive register allocation with a linear number of registers. In *DISC*, volume 8205 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2013.

[DGFGR13] Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Sergio Rajsbaum. Linear space bootstrap communication schemes. In *ICDCN*, volume 7730 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2013.

[FHS98] Faith Ellen Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5) :843–862, 1998.

[Gaf02] Eli Gafni. A simple algorithmic characterization of uniform solvability. In *FOCS*, pages 228–237. IEEE Computer Society, 2002.

[GMT01] Eli Gafni, Michael Merritt, and Gadi Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *PODC*, pages 161–169. ACM, 2001.

[Lam] Leslie Lamport. Proofs for adaptive register allocation with a linear number of registers. http ://research.microsoft.com/en-us/um/people/lamport/tla/snapshot.html.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1) :1–11, February 1987.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lam09] Leslie Lamport. The pluscal algorithm language. In Martin Leucker and Carroll Morgan, editors, *ICTAC*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer, 2009.

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