

# Process Coordination with Fetch-and-Increment

by

*Eric Freudenthal and Allan Gottlieb*

Ultracomputer Note #159

May, 1989

Revised November, 1991

## ABSTRACT

*The fetch-and-add (F&A) operation has been used effectively in a number of process coordination algorithms. In this paper we assess the power of fetch-and-increment (F&I) and fetch-and-decrement (F&D), which we view as restricted forms of F&A in which the only addends permitted are  $\pm 1$ . F&A-based algorithms that use only unit addends are thus trivially expressed with just F&I and F&D. Our primary contributions are new F&I/F&D algorithms for readers/writers coordination including a proof of its correctness and algorithms for barrier synchronization for dynamically-sized groups. We also restructure an existing F&A-based algorithm for queues-with-multiplicity to obtain an algorithm using just F&I and F&D. When executed on certain hardware architectures, most of these algorithms are free of serial bottlenecks. We also discuss a general technique for implementing F&A using F&I/F&D at a cost logarithmic in the number of processors.*

**Key Words and Phrases:** Barrier Synchronization, Bottleneck-free Algorithms, Fetch-and-add, Fetch-and-increment, Parallel Access Queues, Process Coordination, Readers/Writers Problem.

## 1. Introduction

The fetch-and-add instruction, introduced in [GK81], is essentially an indivisible add to memory; its format is  $F\&A(V, e)$ , where  $V$  is an integer variable and  $e$  is an integer expression. This operation is defined to return the *old*<sup>1</sup> value of  $V$  while atomically replacing  $V$  by the sum  $V + e$ . We refer to  $V$  as the *target* and  $e$

---

Both authors received support from the Department of Energy under grant number DE-FG02-88ER25052. The second author received support from the National Science Foundation under grant number MIP-8915488.

A version of this paper without the appendices appeared in the proceedings of the ACM Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 260-268, April 1991.

<sup>1</sup>Note that F&A differs from the earlier ReplaceAdd primitive [GLR83], which returns the *new* value of  $V$ .

as the *addend* of the fetch-and-add. Important special cases occur when the addend is  $\pm 1$ , which are referred to as fetch-and-increment and fetch-and-decrement (F&I and F&D).

F&A has been used in a number of process coordination algorithms, including queues, barriers, semaphores, and shared memory management ([EO88], [GLR83], [Sto82] and [TY90]). As a simple example, note that, if multiple processes execute  $F&I(V)$  (i.e.,  $F&A(V, 1)$ ) concurrently, the values returned are consecutive integers. These values can then be used as indices into an array with the assurance that each array element is assigned to exactly one process. Queue algorithms that are free of critical sections have been developed using this observation.

The present paper studies the following question: Given a F&A-based algorithm, does there exist an equally efficient algorithm using only F&I and F&D? For algorithms in which the only addends used are  $\pm 1$ , such as the queue algorithms suggested above and the P/V and traditional barrier synchronization algorithms discussed below, the answer is trivially yes.

One reason for our interest in this question is that hardware realizations of F&I and F&D are somewhat easier than for F&A, which has led the designers of some systems, e.g. the Stanford DASH multiprocessor [LLG89], to implement F&I/F&D but not F&A. Note, that since an operand need not be sent with F&I, the processor can simply issue a “special” load that the memory subsystem interprets as a F&A. For example, the OPT bits available on the AMD 29000 [AMD89] can be used for this purpose. Finally the hardware technique for combining concurrent fetch-and-add operations on a single-bus multiprocessor discussed in [SSG89] is significantly simpler if the addends are restricted to  $\pm 1$ . (That same paper also notes that the queue algorithms mentioned above use only unit addends.)

The readers/writers problem of Courtois, Heymans, and Parnas [CHP71] has led to an algorithm free of serial bottlenecks that requires F&A with non-unit addends [GLR83]. In this paper, we present a F&I/F&D algorithm that is similarly bottleneck free. A correctness proof appears in Appendix A.

Efficient barrier synchronization is critical for the success of a MIMD shared-memory architecture. We note that an existing F&A-based implementation uses only unit addends. Like many other barriers, this one requires the number of participating processes to be known in advance. To overcome this limitation we present a new F&I-based algorithm for forming groups of processes at run time that may then participate in barrier coordination.

We also present a new implementation of “linear multiqueues”. These queues contain items tagged with multiplicity and permit concurrent processes to obtain copies of an item with no critical sections.

Finally, we consider the general problem of implementing F&A in terms of F&I and F&D and discuss a “software combining” [YTL86] solution having a time complexity logarithmic in the number of processors.

The remainder of this paper is organized as follows. The next section describes our shared-memory MIMD computational model. Sections 3 through 5 discuss semaphores, barriers, and the readers/writers algorithm. Section 6 defines multiqueues and presents algorithms for their implementation. Section 7 briefly describes the simulation of arbitrary F&As using just F&I/F&D. We conclude with some remarks on the relative cost and power of F&I/F&D versus F&A.

## 2. Architectural Considerations

We assume an MIMD shared-memory computer with atomic fetch-and-increment/decrement operations. Although implementing these operations with spinlocks is sufficient for our algorithms to behave correctly, good performance requires that the primitives be implemented efficiently. In an ideal implementation, all memory references, including F&I/F&D, would require a single cycle, even if concurrent references are to the same location. Both synchronous and (weaker) asynchronous models of this behavior have been defined, respectively CRCW PRAMs (see [FW78], [Sni82], [Sch80], and [BH82]) and APRAMs (see [Gib89], [KRS88], and [CZ89]). The algorithms presented below do not assume synchronous execution and thus are correct for either model.

Although this ideal implementation is not realizable since fan-in (and other) limitations preclude single-cycle access to large globally-shared memory, many architects have argued that good engineering

approximations can and have been built:

A shared bus multiprocessor, the most common approximation, necessarily serializes accesses to different memory addresses. However, it is not difficult to combine concurrent loads and stores directed at a single address thereby satisfying them all in a single bus cycle. Furthermore [SSG89] has shown how to extend bus-based combining to include F&I/F&D.

Shared-memory multiprocessors using multistage interconnection networks have also been studied and one, the BBN Butterfly [CGS85], is commercially available and supports fetch-and-add. For such architectures the conflict-free access time to shared memory must grow at least logarithmically with the number of processors. However, the constants are small and even for configurations with thousands of processors, the access times are a few tens of processor cycles, which is not very different from the situation with current vector supercomputers. If the assignment of program variables to memory modules is suitably hashed, contention among accesses to distinct variables will be low. For concurrent accesses to a single variable, combining again helps. BBN [RCC90], IBM [PBG85], and NYU [GGK83] designs have addressed this issue; the last two designs included F&A. Once again, the hardware necessary for supporting (and combining) F&I/F&D is simpler than what is needed for F&A.

### 2.1. Bottleneck-Free Algorithms

Coordination algorithms based on primitives other than F&A (such as test&set or compare&swap), even when run on a CRCW PRAM, have time complexity that grows (usually linearly) with the number of processors involved because these processors must take turns serially executing some (possibly small) critical code section. As available hardware parallelism rises, such *serial bottlenecks* can waste an increasing amount of processing power, and may discourage full exploitation of available concurrency.

In contrast, many F&A algorithms, when run on a CRCW PRAM (or the weaker APRAM with unit-time memory access), have time complexity independent of the requested concurrency. For example, on such a machine, any number of processors can each insert an item onto a single queue in the time required for just one insertion. We refer to these algorithms, which are free of serial code sections, as *bottleneck-free*. Many of the corresponding F&I/F&D algorithms that we offer in the present paper are also bottleneck-free.

Sometimes an algorithm is bottleneck-free for “large enough” problem instances. For example, the linear multiqueues presented in section 6 are bottleneck-free if inserted items have “multiplicity” proportional to the available concurrency. Furthermore, we consider an algorithm bottleneck-free if its only serialization is required by the problem *specification*. For example, consider the readers/writers problem. Writers must execute serially but we classify the implementation below as bottleneck-free since, in the absence of writers, any number of readers execute in the time required for just one.

The *bottleneck-free* property should not be confused with *wait-free* or *non-blocking* as defined in [Her90]. A process participating in a wait-free coordination activity is guaranteed to complete after executing some finite number of steps, even if other processes halt. The weaker non-blocking property guarantees that some (rather than every) non-faulty process makes progress. Both differ from bottleneck-freedom in two important ways: First, wait-free and non-blocking algorithms, even on a CRCW PRAM, may have time complexity that grows with concurrency and second, bottleneck-free algorithms are not required to tolerate a halting processor.

Many of the algorithms presented below are bottleneck-free but none are non-blocking. In contrast, most of the algorithms in [Her90] are non-blocking but none are bottleneck-free. We are aware of only one non-trivial algorithm that has both properties: the consensus algorithm in [Her91] (which is even wait-free). It would be interesting to find more.

### 3. Semaphores

F&A-based semaphores have been implemented in [GLR83]. The algorithms for both binary and counting semaphores use only unit addends and hence are trivially expressible with F&I/F&D (counting semaphores permit up to a pre-specified  $M$  processes to obtain a resource, binary semaphores are the special case  $M = 1$ ). We note that in the absence of contention, efficient implementations of the P and V operations

each generate only one access to shared memory.

[GLR83] also presents PChunk/VChunk algorithms in which multiple units of the resource can be seized or released atomically. These algorithms appear to require F&A with non-unit addends and hence the best we can achieve with F&I/F&D is the logarithmic cost simulation of F&A discussed in section 7.

#### 4. Barrier Synchronization

For the reader's convenience, we present the following F&I/F&D barrier since the dissertation in which it appears [Dim88] has not been widely circulated.

```
constant
  N := number of processes: integer;
shared
  Syncvar := 0 :integer;
procedure Barrier is
var
  WasLess :boolean;
begin
  WasLess := (Syncvar < N);
  if (F&I(Syncvar) = 2*N-1)
    Syncvar := 0;
  while (WasLess = (Syncvar < N));
end.
```

##### 4.1. Dynamic Barriers and Group Formation

Barriers are often used to create a "loosely synchronous" mode of execution. Traditional barriers require that the number of participating processes be known in advance. To overcome this limitation we present a new F&I-based algorithm for forming groups of processes at run time that may then participate in barrier coordination. (Dynamic grouping was introduced in [Dim88], where it was called group lock; see also [LG87].)

Consider an execution of the following code sequence by an arbitrary set  $S$  of processes.

```
GroupProlog
arbitrary application code, a g-section
GroupEpilog
```

A group  $G$  of these processes is formed during prolog execution and maintained through the epilog. Once  $G$  is formed, no process may join. However, a process may leave  $G$  at any time by executing the epilog. Hence the number of processes currently in a formed group is non-increasing with time: A function **GroupSize** is provided to give the current size. Using **GroupSize**, it is not difficult to write a barrier applicable to a  $g$ -section: For example, the simple algorithm presented below suffices. More complicated algorithms which make fewer shared memory accesses have also been developed. Several examples appear in Appendix B.

Our algorithms for **GroupProlog**, **GroupEpilog**, and **GroupSize** are presented in Figure 1. Three shared variables are used:  $T$ , the ticket number, counts the number of executions of **GroupProlog**;  $L$ , the limit value, is the largest ticket number in the current group; and  $E$ , the exit value, counts the number of executions of **GroupEpilog**. **GroupProlog** uses, in addition,  $t$  and  $l$ , which are private versions of  $T$  and  $L$ . The process having the lowest ticket number in a group (which is one plus the limit value of the previous group) is called the group *leader* and is responsible for setting the limit value.

The algorithms are straightforward if one assumes that all variables are mathematical integers, i.e. do not overflow. In a real implementation, overflows can occur. Fortunately, the algorithm as written remains correct under the following assumptions: Two's-complement arithmetic is used in which overflows are ignored (in particular, adding a small positive integer to a very large positive integer produces a negative

integer having very large absolute value), comparisons are performed via subtractions and testing the sign of the result, and the maximum concurrency is less than the maximum (signed) computer integer. The only remaining point to note is that the values of all five program variables ( $T$ ,  $t$ ,  $L$ ,  $l$ , and  $E$ ), when viewed as mathematical integers, can differ from each other by no more than the maximum concurrency.

A simple algorithm for `GroupBarrier` is given in Figure 2. Three shared variables are used:  $C[0..1]$  each count the number of processes that have reached the barrier and  $P$ , the pass number, indicates which count is active. The only subtlety in the code is restricting  $P$  to  $\{0,1\}$  (and hence using only two  $C$ 's). It is not hard to see that each pass through the barrier cleans it for the next pass.

## 5. Readers and Writers

The readers and writers synchronization paradigm, described in [CHP71] coordinates access of two classes of processes to a protected resource. Processes in the first class, *writers*, require exclusive access; whereas, processes in the second class, *readers*, may share the resource with other readers. One often imposes additional *fairness* conditions limiting the time a process may need to wait for access. The most likely cause of unbounded waiting is that readers are permitted to begin while other readers are active and thus a continual stream of readers may starve all writers. We eliminate this possibility by the standard technique of giving writers priority, which naturally does not prevent writers from starving readers. In addition, the implementation below uses an unfair semaphore, which permits writers to starve other writers. These potential starvation have not yet proved troublesome in practice, but our experience to date is limited to small-scale multiprocessors. We have also developed F&I/F&D algorithms for both reader-priority and fair reader/writer coordination.

Many algorithms for readers/writers coordination have appeared; the first in [CHP71], where the problem was introduced. Fetch-and-add based algorithms are given in [GLR83], which, however, make essential use of non-unit addends: readers manipulate a shared variable using F&A with unit addends (essentially P/V on a counting semaphore) but writers use addends  $\pm K$ , for a large constant  $K$  (essentially PChunk/VChunk). The idea is that readers need one unit of a resource, writers need  $K$  units, and there are a total of  $K$  units available. Our algorithm (see Figure 3) avoids non-unit addends by using two separate variables to count the number of readers and writers respectively. The characteristics of this new algorithm are given in Theorem A below:

**Theorem A.** *The reader/writer coordination algorithm has the following properties:*

- a) *Readers exclude writers.*
- b) *Writers exclude other writers.*
- c) *Deadlock is impossible.*
- d) *Readers are not serialized, i.e. their execution is bottleneck-free.*
- e) *Readers can not starve writers.*

The first three items in theorem A are standard safety properties and the fifth is a fairness property common in implementations, such as ours, that enforce writer priority.

The fourth item in Theorem A is somewhat unusual. Assume that no writers are present during a given time interval. Then there exists a (small) constant  $C$  such that any requesting reader will reach its `read` statement (`r6`) after executing at most  $C$  statements. Note that  $C$  does not depend on the number of requesters. To illustrate the strength of this statement, let us assume the CRCW PRAM model of computation. Then, in the absence of writers, any number of requesting readers will all reach their respective `read` statements within  $C$  cycles. That is, the execution is bottleneck-free. To the best of our knowledge only the [GLR83] algorithms, which require non-unit addends, and the algorithm in Figure 3 have this desirable property.

A more formal statement of theorem A with an accompanying proof of correctness appears in Appendix A.

```

shared
  T := 0, L := 0, E := 0: integer;

procedure GroupProlog is
var
  done := false :boolean;
  t,l :integer;
begin
  t := fai(T) + 1;           -- get ticket
  while (not done)         -- loop until granted
  l := L;                   -- read highest ticket # allowed to enter
  if (l ≥ t)                -- am I allowed to enter?
  done := true;             -- ...yes
  else if (l=E and t=l+1)   -- is group done & am I leader of NEXT group
  L := T;                   -- ...yes, form next group
  done := true;
end

procedure GroupEpilog is
begin
  fai(E);                   -- release lock
end

procedure GroupSize :integer is
begin
  GroupSize := L - E;
end

```

**Figure 1. The GroupProlog, GroupEpilog, and GroupSize procedures.**

```

shared
  P := 0: integer;
  C := {0,0}: array [0..1] of integer;

procedure GroupBarrier is
var
  p :integer;
begin
  p := P;                   -- is this an even or odd cycle?
  fai(C[p]);                -- bump the appropriate count
  while (C[p]≠GroupSize and p=P); -- wait for all procs to reach barrier
  P := (p + 1) mod 2;       -- increment pass
  C[p] := 0;                -- reset count
end

```

**Figure 2. The GroupBarrier procedure.**

```

shared
  NumReaders := 0, NumWriters := 0: integer;

procedure Reader is
begin
r1:   while (NumWriters > 0);      -- wait for writer to exit
r2:   f&i(NumReaders);            -- try to get lock
r3:   if (NumWriters > 0)         -- did a writer beat me into the lock?
r4:     f&d(NumReaders);          -- yes; undo increment...
r5:     goto r1;                  -- ...wait, and try again
r6:   read;                       -- Perform reader action
r7:   f&d(NumReaders);            -- release the lock
end Reader

procedure Writer is
begin
w1:   while (NumWriters > 0);     -- only one writer at a time
w2:   if (f&i(NumWriters) > 0)    -- increment numwriters, am I first?
w3:     f&d(NumWriters);          -- no, undo increment...
w4:     goto w1;                  -- ...wait, and try again
w5:   while (NumReaders > 0);    -- wait for readers to exit
w6:   write;                      -- Perform writer action
w7:   f&d(NumWriters);            -- release the lock
end Writer

```

**Figure 3. Readers and Writers algorithm.**

## 6. Multiqueues

Multiqueues are queues supporting the atomic insertion of *multi-items*, i.e. items with multiplicity. This operation, called multi-insert and written  $MInsert(i, m)$  is functionally equivalent to atomically inserting  $m$  instances of item  $i$ . (Note that a multiqueue with two entries, one of multiplicity seven and the other of multiplicity four contains two multi-items and eleven items.) The delete operation removes an item; if this operation depletes the corresponding multi-item (i.e. lowers its remaining multiplicity to zero), the latter is removed as well and the deletion is referred to as a final deletion.

Two families of multiqueue algorithms had been developed previously and named to reflect their worst-case performance: linear multiqueues and logarithmic multiqueues [GLR83]. Before discussing F&I/F&D implementations of multiqueues, we describe circumstances in which linear multiqueues, logarithmic multiqueues, and simple parallel queues (i.e. the queues mentioned in the introduction) are the structures of choice.

Consider a scenario in which a single processor inserts  $m$  identical items, which are subsequently deleted by  $m$  processors concurrently. Using a simple parallel queue, the insertion time is linear in  $m$  (since there are  $m$  inserts performed by a single processor) and the  $m$  deletions all complete in constant time (assuming  $m$  processors are available and using the CRCW PRAM model) so the overall complexity is proportional to  $m$ . Using a linear multiqueue, the original processor would issue a single multi-insert, which takes constant time, and the  $m$  concurrent deletes would again all complete in constant time. Hence, for large  $m$ , the linear multiqueue has removed a linear delay present when using simple queues.

Linear multiqueues have been used in several software systems including a parallel UNIX kernel [ELS88] and a similar structure has been used in a parallel FORTRAN run-time system [Ber88] for scheduling multiple (independent) iterates of a DO loop.

The weakness of linear multiqueues is that, at any given time, only one multi-item can be a deletion site. Now consider a scenario in which  $m$  processors concurrently insert one (distinct) item each, which are subsequently removed by  $m$  concurrent deletes. Using simple queues all the operations finish in constant

```

type
  Multiitem is record
    Item :DataType;
    Remaining := 0 :integer;           -- remaining multiplicity
  end Multiitem
shared
  Head := 0, Tail := 0 :integer;      -- delete and insert indices
  MCount := 0 :integer;              -- count of Multiitems
  Cells: array [0..QSize-1] of Multiitem;
  NumReaders := 0, NumWriters := 0 :integer;  -- for reader-writer coordination

procedure MInsert(Data :DataType, Mult :integer, Full :out boolean) is
var
  Index :integer;
begin
i1:   Full := true;
i2:   if (MCount < QSize)              -- Is queue full?
i3:     if (fai(MCount) < QSize)
i4:       Full := false;
i5:       Index := fai(Tail) mod QSize;  -- get insert site
i6:       Cells[Index].Item := Data;    -- fill in multiitem
i7:       Cells[Index].Remaining := Mult;
i8:     else
i9:       fad(MCount);
end MInsert

procedure Delete(Data :out DataType, Empty :out boolean) is
var
  Index, MyRemaining := 0 :integer;
begin
d1:   Empty := true;
d2:   while (MyRemaining <= 0 and MCount > 0)  -- loop until get an item or Q is empty
d3:     ReaderPrologue;                       -- prevent race conditions (see text)
d4:     if (Cells[Head].Remaining > 1)
d5:       MyRemaining := fad(Cells[Head].Remaining)
d6:     if (MyRemaining > 0)
d7:       Empty := false;
d8:       Data := Cells[Head].Item;
d9:     ReaderEpilogue;
d10:    if (MyRemaining = 1)                  -- did I exhaust multiitem?
d11:      WriterPrologue;                    -- prevent race conditions (see text)
d12:      Head := (Head + 1) mod QSize;
d13:      fad(MCount)                        -- multiitem has been removed
d14:      WriterEpilogue;
end Delete

```

**Figure 4. F&I Algorithms for Linear Multiqueues**

time. Using linear multi-queues the multi-inserts (each of multiplicity 1) all complete in constant time but the  $m$  deletes are directed at distinct multi-items so the time required is linear in  $m$ . Thus the multi-queue has added a linear delay.

In summary: simple queues are good when items have low multiplicity; linear multiqueues are good when items have high multiplicity; and either structure can introduce a linear delay if used where the other is favored. The final possibility is to employ logarithmic multiqueues. Using this last (tree-like) structure, all operations can be performed concurrently but each has complexity logarithmic in  $QSize$ , the size of the multiqueue, i.e. in the maximum number of multi-items that can be stored. Thus, both previous scenarios

have logarithmic complexity when these structures are employed. Logarithmic multiqueues might be appropriate if the expected item multiplicity is unknown.

A recent implementation of logarithmic multiqueues [Dim88] uses only one F&A with non-unit addend. Simulating this operation with F&I as described in section 7, adds a complexity term logarithmic in the number of processors. For most applications, this term is dominated by the basic  $\log(\text{QSize})$  cost of logarithmic multiqueues.

### 6.1. Linear Multiqueues

Linear multiqueues utilize a bottleneck-free queue (such as the one suggested in the introduction) to store *multi-items* (see Figure 4). Previous algorithms for these structures maintain counts of both multi-items and items and update the latter by  $m$  to reflect an insertion with multiplicity  $m$ . This update is the only F&A with non-unit addend used. However, we have observed that the algorithms can be restructured to eliminate the item count. The result, shown in Figure 4, requires only F&I/F&D.

We now describe the `MInsert` and `Delete` algorithms, which respectively add a multi-item to the tail of the multiqueue and lower the multiplicity of the head multi-item (discarding the latter if its multiplicity reaches zero).

`MInsert` begins by checking for a full queue. Although the tests in `i2` and `i3` may appear redundant, [GLR83] illustrates race conditions that can occur if this so called test-increment-retest (TIR) sequence is shortened. If the multiqueue was full, the insert fails and the multi-item count is restored, if necessary, at `i9`. If the multiqueue was not full, the count is incremented by the TIR, `Tail` is then updated to determine the insert site, and the multi-item is filled in.

It is possible that the F&I performed on line `i5` causes `Tail` to overflow. One solution, applicable on machines in which overflow silently reduces the result by a value  $V$ , is to choose `QSize` a divisor of  $V$ . For example on many current computers,  $V$  is a power of two and we may choose `QSize` to be any smaller power of two. Alternatively one can modify the algorithm to prevent overflows.

The loop `d2..d9` in `Delete` attempts to remove an item from the head multi-item, which is the deletion site (all concurrent deletions have the same site). Each attempt is accomplished by lowering the multi-item's remaining multiplicity and is repeated until either the multiqueue is empty (`MICount = 0`) or the removal succeeds (`MyRemaining > 0`). An empty multiqueue causes the deletion to fail. We first describe the actions taken by a successful deletion and then discuss how potential race conditions are avoided. A successful test-decrement-retest (TDR) sequence `d4..d6` implies that the current head multi-item is non-empty and decrements its multiplicity to reflect the deletion in progress. A copy of the item is then placed in an output parameter. If the TDR at `d10` indicates that the deletion site has just been exhausted, the current operation is called a *final deletion* and the subsequent multi-item is made the new head. An unsuccessful TDR results in the loop being repeated.

The apparently redundant test (line `d4`) in the delete TDR prevents `Cells[Index].Remaining` from underflowing past `MinInt` to a positive value, which would lead to erroneous behavior. Note that one need not undo the decrement performed by an unsuccessful TDR since the multi-item is empty and the final deletion (already in progress) will make a new multi-item the head.

Note two points. First, readers/writers coordination<sup>2</sup> guarantees that the head pointer is not modified by a final deleter while another deleter has a copy. Second, the ordering of shared accesses ensures that a multi-item's value is valid before it is read: inserters write the value **before** the multiplicity; deleters test the multiplicity (at `d5..d6`) **before** reading the value.

---

<sup>2</sup>The `ReaderPrologue` in Figure 4 corresponds to lines `r1..r5` of the `Reader` program in Figure 3. Similarly `ReaderEpilogue` corresponds to `r7`, `WriterPrologue` corresponds to `w1..w5`, and `WriterEpilogue` corresponds to `w7`.

## 7. Simulating F&A with F&I using Software Combining

F&A can easily be simulated by a load-add-store sequence enclosed in a semaphore to guarantee atomicity. This simple implementation serializes concurrent F&As and hence gives an execution time linear in the number of processors that issue a F&A simultaneously. This serial bottleneck can be reduced to time logarithmic in the number of processors by using a technique known as software combining [YTL86]. The logarithmic cost software combining algorithm presented in [GVW89] requires only binary semaphores for its implementation. Since binary semaphores can themselves be implemented using only F&I/F&D, we can adapt the software combining algorithm of [GVW89] to obtain a logarithmic cost implementation of F&A using only F&I/F&D.

## 8. Conclusion

We have demonstrated that several important fetch-and-add based coordination algorithms can be implemented as efficiently using just fetch-and-increment and fetch-and-decrement. For some algorithms, such as simple queues, standard barriers, and semaphores, the result is trivial; only unit addends are used in the F&A version. For linear multiqueues, we have observed that the only F&A with non-unit addend can be eliminated without increasing the computational complexity. Finally, our main contributions are F&I/F&D algorithms for the readers/writers problem and for a barrier coordinating a variable number of processes.

Most of these algorithms, like their older F&A counterpart, are bottleneck-free. For the problems considered, we know of no other algorithms with this property.

For algorithms in which F&As with non-unit addends are necessary, we have observed that the technique of software combining enables one to simulate any F&A in logarithmic time using F&I/F&D.

We remain advocates of parallel computers with hardware support for F&A with arbitrary addends (as well as for other fetch-and-ops) but are aware that the results presented show that the incremental gain of F&A over F&I/F&D is less than we had previously suspected. In some architectures the incremental cost of supporting F&A over F&I/F&D may be significant. For example, combining concurrent F&As on a single-bus multiprocessor is much easier if the addends are limited to  $\pm 1$ . Designers of new parallel architectures should weigh the relative simplicity of F&I against the resulting algorithmic tradeoffs.

## Acknowledgment

We thank our colleagues at the NYU Ultracomputer Research Laboratory, especially Jan Edler, Malcolm Harrison, Cong Ly, and David Wood, for helpful discussions concerning the algorithms presented in this paper. We also thank Susan Dickey, Richard Kenner, and the referees for improving our earlier drafts. Finally, we thank Richard Cole and Dennis Shasha for sharing with us their expertise in asynchronous PRAMS and wait-free algorithms, respectively.

## Bibliography

- [AMD89] Advanced Micro Devices, *Am29000 User's Manual*, Sunnyvale, California, 1989.
- [Ber88] Wayne Berke, "ParFOR—A Structured Environment for Parallel FORTRAN", Ultracomputer Note #137, Courant Institute, New York, NY, 1988.
- [BH82] Alan Borodin and John E. Hopcroft, "Routing, Merging, and Sorting on Parallel Models of Computation", *14th Annual ACM Symposium on Theory of Computing*, pp. 338-344, 1982.
- [CGS85] William Crowther, John Goodhue, Edward Starr, Robert Thomas, Walter Milliken, and Tom Blackadar, "Performance Measurements on a 128-Node Butterfly Parallel Processor", *Proc ICCP*, pp. 531-540, 1985.
- [CHP71] P. Courtois, F. Heymans, and D. Parnas, "Concurrent Control with 'Readers' and 'Writers'", *CACM* **14** Number 10, pp. 667-668, Oct. 1971.
- [CZ89] Richard Cole and Ofer Zajicek, "The APRAM: Incorporating Asynchrony into the PRAM Model" "A More Practical PRAM Model", *Proceedings of the 1989 Symposium on Parallel Algorithms and Architectures*, pp. 169-178, June 18-21, Santa Fe, New Mexico.
- [Dim88] Isaac Dimitrovsky, "ZLISP—A Portable Parallel LISP Environment", Doctoral Dissertation, Courant Institute, New York, NY, 1988.
- [ELS88] Jan Edler, Jim Lipkis, and Edith Schonberg, "Process Management for Highly Parallel UNIX Systems", *Proc. Usenix Workshop on UNIX and Supercomputers*, Pittsburgh, 1988a, pp. 1-17.
- [EO88] Carla Schlatter Ellis and Thomas J. Olson, "Algorithms for Parallel Memory Allocation", *Intern. J. of Parallel Programming* **17** Number 4, pp. 303-345, August 1988.
- [FG91] Eric Freudenthal and Allan Gottlieb, "Process Coordination with Fetch-and-increment", NYU Ultracomputer Note 159, New York University, revised 1991 (an expanded version of the present paper).
- [FW78] Steven Fortune and James Wylie, "Parallelism in Random Access Machines", *Proc. 10th ACM Symp. on Theory of Computation*, pp. 114-118, 1978.
- [GGK83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Lawrence Rudolph, and Marc Snir, "The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer", *IEEE Trans. Comp.*, pp. 175-189, February 1983.
- [Gib89] Phillip B. Gibbons "A More Practical PRAM Model", *Proceedings of the 1989 Symposium on Parallel Algorithms and Architectures*, pp. 158-168, June 18-21, Santa Fe, New Mexico.
- [GK81] Allan Gottlieb and Clyde P. Kruskal, "Coordinating Parallel Processors: A Partial Unification", *Computer Architecture News*, pp. 16-24, October 1981.
- [GLR83] Allan Gottlieb, Boris Lubachevsky, and Larry Rudolph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors", *ACM TOPLAS* **5** pp. 164-189, April 1983. Multiqueues appear only in the NYU technical report.
- [GVW89] James R. Goodman, Mary K. Vernon, and Phillip J. Woest, "Efficient Synchronization Primitives for Large-scale Cache-coherent Multiprocessors", *Proc. ASPLOS III*, pp. 64-75, April 1989.
- [Her90] Maurice P. Herlihy, "A methodology for implementing highly concurrent data structures", *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 197-206, March 1990.
- [Her91] Maurice P. Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems* **13**, Number 1, January 1991.
- [KRS88] Clyde P. Kruskal, Larry Rudolph, and Marc Snir, "A Complexity Theory of Efficient Parallel Algorithms", *Proc. 15th ICALP*, pp. 333-346, 1988.

- [LG87] Boris Lubachevsky and Albert Greenberg, "Simple, Efficient Asynchronous Parallel Prefix Algorithms", *Proc. ICCP*, pp. 67-69, Aug 1987.
- [LLG89] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam, "Design of the Stanford DASH Multiprocessor"; Technical Report: CSL-TR-89-403, Computer Systems Laboratory, Stanford University, December 1989.
- [PBG85] Gregory F. Pfister, William C. Brantley, David A. George, Steve L. Harvey, Wally J. Kleinfi elder, Kevin P. McAuliffe, Evelin S. Melton, V. Alan Norton, and Jodi Weiss, "The IBM Research Parallel Processor Prototype (RP3): "Introduction and Architecture" , *Proc. ICCP*, pp. 764-771, Aug. 1985.
- [RCC90] Randall D. Rettberg, William R. Crowther, Phillip P. Carvey, "The Monarch Parallel Processor Hardware Design", *IEEE Computer*, pp. 18-30, April 1990.
- [Sch80] Jacob T. Schwartz, "Ultracomputers", *ACM TOPLAS* 2 Number 4, pp. 484-521, October 1980.
- [Sni82] Marc. Snir, "On Parallel Search", *Proc. Principles of Distributed Computing*, pp. 242-253, Aug 1982.
- [Sto82] Harold S. Stone, "Parallel Memory Allocation Using the Fetch-and-Add Instruction", Technical Report RC9674, IBM Thomas J. Watson Research Center, November 1982.
- [SSG89] Gurindar S. Sohi, James Smith, and James R. Goodman, "Restricted Fetch and  $\phi$  Operations for Parallel Processing", *Intern. Conf. on Supercomputing*, Harklion-Crete, Greece, June, 1989.
- [TY90] Peiyi Tang, and Pen-Chung Yew, "Algorithms for Hot-Spot Addressing", *J. of Parallel and Distributed Computing*, to appear.
- [YTL86] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie, "Distributing Hot-spot Addressing in Large-Scale Multiprocessors", *Proc. ICCP*, 1986.

## Appendix A: Correctness Proof for Readers-Writers Algorithm

The looping programs for Reader and Writer which appear in figure 5 contain the readers/writers algorithms of figure 2 which, to simplify our proof, are encoded at a lower level. The following proof refers to this version of the algorithms.

We assume a finite set of reader processors, i.e. processors continually executing the reader program, and a finite set of writer processors. We assume further that at time zero, the start of execution, all processors are at the first statement of their respective programs and note that at this point  $\text{NumReaders}=\text{NumWriters}=0$ . As indicated in section 2, we use (instead of a PRAM) a computational model in which, during each time unit, exactly one (rather than every) processor executes one statement of its program.<sup>3</sup> Finally, we assume that processors are not starved, i.e. during any infinite time interval, each processor executes infinitely often. We use the symbol  $\square$  to indicate the end of a proof.

To state our theorem precisely, we need the notions of state and history. The *state* of an execution consists of the values of the variables and program counters. In fact, we show below that the values of

```
Shared variables:
NumReaders := 0, NumWriters := 0: integer;

program Reader is
begin
r0:   non-reader-writer;           -- arbitrary code outside R/W section
r1:   if (NumWriters > 0) goto r1;  -- wait for writer to exit
r2:   f&i(NumReaders);             -- try to get lock
r3:   if (NumWriters = 0) goto r6;  -- did a writer beat me into the lock?
r4:   f&d(NumReaders);             -- yes; undo increment...
r5:   goto r1;                    -- ...wait, and try again
r6:   read;                       -- Perform reader action
r7:   f&d(NumReaders);             -- release the lock
r8:   goto r0
end Reader

program Writer is
begin
w0:   non-reader-writer;           -- arbitrary code outside R/W section
w1:   if (NumWriters > 0) goto w1;  -- only one writer at a time
w2:   if (f&i(NumWriters) = 0) goto w5; -- increment numwriters, am I first?
w3:   f&d(NumWriters);             -- no, undo increment...
w4:   goto w1;                    -- ...wait, and try again
w5:   if (NumReaders > 0) goto w5;  -- wait for readers to exit
w6:   write;                       -- Perform writer action
w7:   f&d(NumWriters);             -- release the lock
w8:   goto w0
end Writer
```

**Figure 5. Looping Readers and Writers Programs for Proof.**

---

<sup>3</sup>This assumption is stronger than necessary; we make it for pedagogical reasons. For example, we need not require that the statements **read** and **write** take only one time unit each to execute; it suffices that each execution of **read** terminates in finite time. If one permits a non-terminating **read**, then writers can be starved, violating part *e* of the theorem. The special statement **non-reader-writer** represents an arbitrary code sequence modifying neither  $\text{NumReaders}$  nor  $\text{NumWriters}$ . We need no restrictions on the time required to execute **non-reader-writer**; in particular, it need not terminate. Note that the remaining statements are written at essentially the assembly language level so it is reasonable to assume that each executes atomically. In particular, none of these statements makes more than one reference to shared memory. We consider F&I one reference; simply place the increment logic at the memory.

NumReaders and NumWriters, the only variables present, are determined by the program counters and hence are redundant. An *execution history*, or *history*, is a sequence of states  $(s_0, s_1, \dots)$  with  $s_i$  the state of the system at time  $i$ . That is,  $s_0$  is the initial state and  $s_{i+1}$  results from  $s_i$  by having exactly one processor execute its next statement.

We use the following definitions in the proof of theorem A.

$R$  is a finite set of reader processors.

$W$  is a finite set of writer processors.

$\emptyset$  is the empty set.

$NumWriters(t)$

is the value of NumWriters at time  $t$ .

$NumReaders(t)$

is the value of NumReaders at time  $t$ .

$r_\rho(t)$  is the net contribution of  $\rho \in R$  to NumReaders at time  $t$ , i.e. the number of executions of  $f\&i(NumReaders)$  by  $\rho$  up to time  $t$  minus the number of executions of  $f\&d(NumReaders)$  by  $\rho$  up to time  $t$ .

$w_\omega(t)$  is the analogous net contribution of  $\omega \in W$  to NumWriters.

$Steps_\pi(t_1, t_2)$

is the number of statements processor  $\pi$  executes between times  $t_1$  and  $t_2$ .

$L_\pi(t)$  is the label of the next statement to be executed by processor  $\pi$  at time  $t$ .

$last_\pi(l, t)$  is the most recent time prior to  $t$  that processor  $\pi$  executed the statement labeled  $l$ , i.e.  $\max \{i < t: L_\pi(i) = l \text{ and } Steps_\pi(i, i+1) = 1\}$ .

$InRange(t, n, m)$

is the set of processors for which, at time  $t$ ,  $n \leq L(t) \leq m$ , where  $n$  and  $m$  both are statement labels in the same program and are ordered according to their occurrence in the program.

$R_{req}(t)$  is the set of processors that are requesting to read at time  $t$ . More precisely,  $R_{req}(t) = InRange(t, r1, r5)$ .

$R_{read}(t)$  is the set of processors that are reading at time  $t$ . More precisely,  $R_{read}(t) = InRange(t, r6, r6)$ .

$W_{req}(t)$  is the set of processors that are requesting to write at time  $t$ . More precisely,  $W_{req}(t) = InRange(t, w1, w5)$ .

$W_{write}(t)$  is the set of processors that are writing at time  $t$ . More precisely,  $W_{write}(t) = InRange(t, w6, w6)$ .

$W_{active}(t)$  is the set of writers that are not at statement  $w0$  at time  $t$ . More precisely,  $W_{active}(t) = InRange(t, w1, w8)$ .

$Requesting(t)$

is the set of processors that are requesting to read or requesting to write at time  $t$ . More precisely,  $Requesting(t) = R_{req}(t) \cup W_{req}(t)$ .

Using the notation just introduced, we can now state and prove the formal version of our theorem.

**Theorem A (formal version).** Any execution history of the F&I implementation of readers and writers satisfies:

- a) There does not exist a time  $t$  and processors  $\rho \in R$  and  $\omega \in W$  such that  $L_\rho(t) = r6$  and  $L_\omega(t) = w6$ .
- b) There does not exist a time  $t$  and processors  $\omega_1, \omega_2 \in W$  such that  $L_{\omega_1}(t) = L_{\omega_2}(t) = w6$ .
- c) For any time  $t$ , if  $\text{Requesting}(t) \neq \phi$ , there exists a time  $t' \geq t$  such that  $R_{\text{read}}(t') \cup W_{\text{write}}(t') \neq \phi$ .
- d) There exists a constant  $K$  ( $K = 5$  suffices) such that for any reader  $\rho$  and times  $t_1 < t_2$ : If  $\rho \in R_{\text{req}}(t_1)$ ,  $W_{\text{active}}(t') = \phi$  for all  $t_1 \leq t' \leq t_2$  and  $\text{Steps}_\rho(t_1, t_2) \geq K$ , then  $\rho \in R_{\text{read}}(t)$  for some time  $t_1 \leq t \leq t_2$ .
- e) For any time  $t$  such that  $W_{\text{req}}(t) \neq \phi$ , there exists  $t' \geq t$  such that  $W_{\text{write}}(t') \neq \phi$ .

**Proof of theorem A.** We define the *label sequence* of a processor  $\pi$  to be the sequence of labels of the statements executed by  $\pi$ . A simple analysis of Reader and Writer as sequential programs yields the following two propositions.

**Proposition 1.** The label sequence of a single reader processor must satisfy the regular expression

$$(r0\ r1\ r1^*\ (r2\ r3\ r4\ r5\ r1\ r1^*)^*\ r2\ r3\ r6\ r7\ r8)^*$$

Similarly, the label sequence of a single writer processor must satisfy the regular expression

$$(w0\ w1\ w1^*\ w2\ (w3\ w4\ w1\ w1^*\ w2)^*\ w5\ w5^*\ w6\ w7\ w8)^*\ \square$$

**Proposition 2.** For all times  $t$  and reader processors  $\rho$ ,  $r_\rho(t)$  is either 0 or 1, specifically,  $r_\rho(t)$  is 1 if and only if  $L_i(t) \in \{r3, r4, r6, r7\}$ . Likewise, for all times  $t$  and writer processors  $\omega$ ,  $w_\omega(t)$  is either 0 or 1, specifically 1 if and only if  $L_i(t) \in \{w3, w5, w6, w7\}$ .  $\square$

Since clearly  $\text{NumReaders}(t) = \sum r_\rho(t)$  and  $\text{NumWriters}(t) = \sum w_\omega(t)$ , the following result follows easily from Proposition 2.

**Proposition 3.** For all times  $t$ ,  $\text{NumReaders}(t) \geq 0$ .  $\text{NumReaders}(t) > 0$  iff there exists a reader  $\rho \in R$  such that  $L_\rho(t) \in \{r3, r4, r6, r7\}$ . Similarly,  $\text{NumWriters}(t) \geq 0$ .  $\text{NumWriters}(t) > 0$  iff there exists a writer  $\omega \in W$  such that  $L_\omega(t) \in \{w3, w5, w6, w7\}$ .  $\square$

Combining propositions 1 and 3 we obtain the following result, which asserts that  $\text{NumReaders}$  (resp.  $\text{NumWriters}$ ) is positive throughout a non-trivial time interval preceding and containing each read (resp. write).

**Proposition 4.** For any reader  $\rho$ , and time  $T_{r6}$  such that  $L_\rho(T_{r6}) = r6$ , we have  $r_\rho(t) = 1$  for all  $t$  satisfying  $\text{last}_\rho(r2, T_{r6}) < t \leq T_{r6}$  and hence, for the same range of  $t$ ,  $\text{NumReaders}(t) > 0$ . For any writer  $\omega$ , and time  $T_{w7}$  such that  $L_\omega(T_{w7}) = w7$ , we have  $w_\omega(t) = 1$  for all  $t$  satisfying  $\text{last}_\omega(w2, T_{w7}) < t \leq T_{w7}$ , and hence, for the same range of  $t$ ,  $\text{NumWriters}(t) > 0$ .  $\square$

We now prove part a of theorem 1, that readers and writers can not be active simultaneously. Assume the contrary, that there exist a time  $T$  with a reader  $\rho \in R_{\text{read}}(T)$  and a writer  $\omega \in W_{\text{write}}(T)$ .

Let  $t_{r2} = \text{last}_\rho(r2, T)$  and  $t_{r3} = \text{last}_\rho(r3, T)$ . From proposition 1,  $t_{r2} < t_{r3} < T$ . From inspection of Reader as a serial program,  $\text{NumWriters}(t_{r3}) = 0$ . Finally, from proposition 4,  $\text{NumReaders}(t) > 0$  for all  $t_{r2} < t \leq T$ .

Similarly, Let  $t_{w2} = \text{last}_\omega(w2, T)$  and  $t_{w5} = \text{last}_\omega(w5, T)$ . From proposition 1,  $t_{w2} < t_{w5} < T$ . From inspection of Writer as a serial program,  $\text{NumReaders}(t_{w5}) = 0$ . Finally, from proposition 4,  $\text{NumWriters}(t) > 0$  for all  $t_{w2} < t \leq T$ .

But,  $t_{r3} < t_{w2}$  because  $\text{NumReaders}(t_{w5}) = 0$ ,  $t_{w5} < t_{r2}$  and,  $\text{NumWriters}(t_{r3}) = 0$ . From this, we obtain the contradiction  $t_{r3} < t_{w2} < t_{w5} < t_{r2} < t_{r3}$ , which proves part a.  $\square$

Part b, the mutual exclusion of writers, holds because statements  $w5$  and  $w6$  of Writer are protected with code equivalent to the binary semaphore algorithm proved correct in [GLR83].  $\square$

We now prove part d (readers are not serialized) by showing that, in the absence of writers, any requesting reader  $\rho$  will read after executing at most 5 statements. More precisely, let  $\rho \in R_{\text{req}}(t_1)$  and choose  $t_2$  such that  $\text{Steps}_\rho(t_1, t_2) \geq 5$  and  $W_{\text{active}}(t) = 0$  for all  $t_1 < t < t_2$ . Then part d reduces to:

**Proposition 5.** *There exists a  $t$  in the range  $t_1 \leq t \leq t_2$  such that  $L_\rho(t) = r6$ .*

From proposition 3,  $NumWriters(t) = 0$  for all  $t_1 \leq t \leq t_2$ , which converts  $r1$  to a nop and  $r3$  to a goto. With these conversions, the label sequence of  $\rho$ , during the time interval from  $t_1$  to  $t_2$ , must be a contiguous subsequence of (a sequence satisfying) the regular expression

$$r4 r5 r1 r2 r3 r6 r7 r8 (r0 r1 r2 r3 r6 r7 r8)^*$$

Since  $L_\rho(t_1) \in \{r1, r2, r3, r4, r5\}$ ,  $\rho$  will execute statement  $r6$  (read) after no more than five transitions, completing the proof of proposition 5 and hence the proof of part *d* of theorem A.  $\square$

We now prove part *e*: if  $W_{req}(T) \neq \phi$ , there exists a time  $T' \geq T$  such that  $W_{write}(T') \neq \phi$ . The writer program is equivalent to a binary semaphore protecting the critical section  $\{w5, w6\}$ . The deadlock freedom of this semaphore implies there exists a writer  $\omega$  and time  $t \geq T$  such that  $L_\omega(t) \in \{w5, w6\}$ . If  $L_\omega(t) = w6$ , we are done. We assume instead that  $L_\omega(t') = w5$  for all  $t' > t$  (since the only other successor to  $w5$  is  $w6$ ). From proposition 3,  $NumWriters(t') > 0$  for  $t' > t$ , which converts  $r3$  to a nop and  $r1$  to a nonterminating loop. Hence, the label sequence of each reader starting at time  $t$  must be a contiguous subsequence of

$$((r6 r7 r8 r0) | (r2 r3 r4 r5)) r1^*$$

Since all processors make non-zero progress, there exists a time  $t_{noR} > t$  such that for all  $t' > t_{noR}$  and  $\rho \in R$ ,  $L_\rho(t') \in \{r0, r1\}$ , and hence (from proposition 3),  $NumReaders(t') = 0$ . Therefore, for all  $t' > t_{noR}$ ,  $w5$  is converted to a nop which forces  $\omega$  to execute  $w6$ , completing the proof of part *d*.  $\square$

We conclude the proof of theorem A by showing that part *c* can be deduced from parts *d* and *e*. Choose a time  $t$  such that  $Requesting(t) \neq \phi$ . We need to find  $t' \geq t$  such that  $R_{read}(t') \cup W_{write}(t') \neq \phi$ . The idea is that a requesting writer will write due to part *e* and if no writers are requesting, a requesting reader will read due to part *d*.

More formally, we begin by noting that if there exists  $\omega \in InRange(t, w6, w6)$  the result is trivial (let  $t' = t$ ). In addition, if there exists  $\omega \in InRange(t, w1, w5)$ , the result follows from part *e*. Hence we may assume that for all  $\omega \in W$ ,  $L_\omega(t) = \{w0, w7, w8\}$ .

Next observe that we may extend the last assumption to all  $t_1 > t$  since if  $InRange(t_1, w1, w6) \neq \phi$ , the result again follows from part *e*. But the only successor to  $w7$  is  $w8$ , the only successor to  $w8$  is  $w0$  and all processors make non-zero progress. Therefore, there exists a time  $t_{noW} \geq t$  such that for all  $t_2 > t_{noW}$ ,  $InRange(t_2, w1, w8) = \phi$ .

We now return to time  $t$  and observe that since  $W_{req}(t) = \phi$  and  $Requesting(t) \neq \phi$ , we can find  $\rho \in R_{req}(t)$ . But from proposition 1, either there exists a  $t'$  in the range  $t \leq t' \leq t_{noW}$  such that  $\rho \in R_{read}(t')$  and we are done, or  $\rho \in R_{req}(t_{noW})$  and the result follows from part *d*.  $\square$

## Appendix B. More GroupBarrier Algorithms

In this appendix, we present three additional algorithms for dynamic barriers. Two are replacements for the original algorithm in figure 2; they make fewer shared memory accesses (especially in the important special case of singleton groups) but are harder to understand. The third has the fewest shared memory accesses but requires an extra assumption. All three save one shared reference by storing locally the value returned by the initial `fail(C[p])`.

The key difference between the first two algorithms is that, in the first, processes increment and poll a single variable `C[p]`; whereas, in the second, processes increment `C[p]` and poll `P`. The one variable solution is more direct and hence in principle faster but, unless combining is provided, can lead to severe memory contention, especially on machines with coherent caches that *invalidate* entries on writes.

If all processes in a group are *guaranteed* to execute the same number of barriers: `GroupSize` need be invoked only once by each process prior to the execution of the first barrier, which leads to the third algorithm, in which `size` is passed as a parameter.

```

procedure GroupBarrierA is
var
  p, s, c :integer;
begin
a1   s := GroupSize();
a2   if (s = 1)
a3     return;
a4   p := P;
a5   c := fai(C[p]) + 1;
a6   while (c≠s)
a7     c := C[p];
a8     if (c=0)
a9       P := !p;
a10    return;
a11    else if (c=s)
a12      break;
a13    s := GroupSize();
a14    C[p] := 0;
a15    P := !p;
end

```

*-- If I am the only process in this group...*  
*-- ...there is no need to synchronize!*  
*-- Is this an even or odd cycle?*  
*-- Bump **and read** the appropriate count.*  
*-- Loop while barrier not satisfied...*  
*-- Read barrier count.*  
*-- did someone else reset the count?*  
*-- Change phase.*  
*-- done.*  
*-- Is barrier satisfied?*  
*-- Group size may decrease!*  
*-- Reset count.*  
*-- Change phase.*

```

procedure GroupBarrierB is
var
  p, s, c :integer;
begin
b1   s := GroupSize();
b2   if (s = 1)
b3     return;
b4   p := P;
b5   c = fai(C[p])
b6   if (c = 0)
b7     while (C[p] ≠ s)
b8       s := GroupSize();
b9       P := !p;
b10      C[p] := 0;
b11   else
b12     while (p = P)
b13     ;
end

```

*-- If I am the only process in this group...*  
*-- ...there is no need to synchronize!*  
*-- Is this an even or odd cycle?*  
*-- Bump the appropriate count.*  
*-- First process only...*  
*-- Loop while barrier not satisfied.*  
*-- Group size may decrease!*  
*-- Change phase.*  
*-- Reset count; done.*  
*-- Rest of processes...*  
*-- Wait until phase changes; done.*

```

procedure GroupBarrierC(size :integer) is
var
  p, s, c :integer;
begin
c1   p := P;
c2   c := fai(C[p]) + 1;
c3   while (c≠size)
c4     c := C[p];
c5     if (c=0)
c6       P := !p;
c7     return;
c8   C[p] := 0;
c9   P := !p;
end

```

*-- Is this an even or odd cycle?*  
*-- Bump **and read** the appropriate count.*  
*-- Loop while barrier not satisfied...*  
*-- Read barrier count.*  
*-- did someone else reset the count?*  
*-- Change phase.*  
*-- done.*  
*-- Reset count.*  
*-- Change phase.*