Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors

by

Allan Gottlieb, B.D. Lubachevsky, and Larry Rudolph†

Ultracomputer Note #16
December 1980
revised December 1981

† This work was supported in part by the Applied Mathematical Sciences Program of the U.S. Department of Energy under Contract No. DE-AC02-76ERO3077, and in part by the National Science Foundation under Grant No. NSF-MCS79-21258.

This report (minus sections 2.3, 8, and 9) appeared in the April 1983 issue of **TOPLAS.**

ABSTRACT

In this paper we implement several basic operating system primitives by using a ‘‘replace-add’’ operation, which can supersede the standard ‘‘test and set’’, and which appears to be a universal primitive for efficiently coordinating large numbers of independently acting sequential processors. We also present a hardware implementation of replace-add that permits multiple replace-adds to be processed nearly as efficiently as loads and stores. Moreover, the crucial special case of concurrent replace-adds updating the same variable is handled particularly well: If every PE simultaneously addresses a replace-add at the same variable, all these requests are satisfied in the time required to process just one request.

1. Introduction

Very large scale parallel processing, made possible by the refinement of VLSI technology, is becoming a reality. Although current MIMD (multiple instruction streams - multiple data streams) configurations rarely include more than a few dozen processing elements (PEs), much larger configurations are being designed (Burroughs [79], CHoPP (see Sullivan et al. [77]), NYU Ultracomputer (see Gottlieb et al. [81]), etc.) and configurations involving tens of thousands of PEs will soon be feasible.

Since in such configurations the relative cost of serial bottlenecks rises linearly with the number of PEs present, users of these future ultra-large scale parallel machines will be anxious to avoid the use of critical (and hence necessarily serial) code sections, even if these sections are short enough to be entirely acceptable in current practice.

In this paper we implement several basic operating system primitives by using a ‘‘replace-add’’ operation, which can supersede the standard ‘‘test and set’’, and which appears to be a universal primitive for efficiently coordinating large numbers of independently acting sequential processors. We also present a hardware implementation of replace-add that permits multiple replace-adds to be processed nearly as efficiently as loads and stores. Moreover, the crucial special case of concurrent replace-adds updating the same variable is handled particularly well: If every PE simultaneously addresses a replace-add at the same variable, all these requests are satisfied in the time required to process just one request.

Critical sections, used to enforce mutual exclusion when multiprocessing a single PE, were introduced by Dijkstra [65] and later refined by Knuth [66] and Eisenberg and McGuire [72]. Later, Dijkstra [74] and Lamport [74] studied similar issues for parallel processing. Solutions to these problems were considered in the context of possible PE malfunctions by Rivest and Pratt [76], Peterson and Fischer [77], Katseff [78], and Holober and Snyder [79]. Although our paper considers similar issues, we assume a somewhat different computational model and do not directly address the problem of malfunctioning PEs. Various multiprocessor synchronization primitives, including those used in this paper, have been compared by Lipton [74], Burns et al. [78], Henderson and Zalcstein [78], and Dolev [79].

The paper is organized as follows. First, our ‘‘paracomputer’’ model of computation is explained and the replace-add operation is defined (section 2). We then use the replace-add operation to implement semaphores (section 3) and to solve the readers/writers problem without recourse to critical section code (section 5). A distributed queue management technique that also avoids the use of critical sections is derived (section 6) and then enhanced to form the core of a distributed operating system scheduler (section 8). Some of our experiences applying these concepts to parallel scientific application codes are discussed (sections 7 and 10). Finally, the replace-add hardware design is outlined (section 12) and appendix A presents a proof of correctness for our semaphore implementation.

2. Computational Model

The replace-add operation, on which many of our considerations are based, was introduced in the 1967 studies of the Athene hypothetical parallel computer system (Draughon et al. [67]). Before describing this operation, a generalized test and set that appears to be an attractive primitive for coordinating concurrent processes, we first discuss our model of parallel computation.

2.1. The Machine

An ideal parallel processor, dubbed a ‘‘paracomputer’’ by Schwartz [80a], consists of identical PEs sharing a common memory. The individual PEs may also have attached local memory, which we refer to as their ‘‘private’’ memories; the memory shared by and common to all processors is called ‘‘public’’, and variables stored there are called ‘‘public variables’’. The PEs can simultaneously read any public cell in one cycle. Moreover, simultaneous writes (including the replace-add operation described below) are likewise effected in a single cycle and a memory cell to which such writes are directed will contain some one of the quantities written into it. This requirement on simultaneous memory updates illustrates the (paracomputer) serialization principle: The effect of simultaneous actions by the PEs is as if the actions occurred in some (unspecified) serial order. Note that simultaneous memory updates are *not* serialized; in fact they are accomplished in one cycle. The serialization principle speaks only of the effect of their action and not of their implementation. (Paracomputers must be regarded as idealized computational models since physical fan-in limitations prevent their realization.)

Our (realizable) approximation to a paracomputer is an MIMD parallel processor in which each PE can directly access its private memory and can access the public memory via a (multicycle) interconnection network. Since in this more realistic architecture a public memory access may require many PE cycles, we must carefully define the notion of simultaneity: Two actions r1 and r2 are *simultaneous* if r1 starts before r2 finishes and r2 starts before r1 finishes.

2.2. Replace-Add

The format of the replace-add operation, which forms the basis of much of our subsequent discussion, is RepAdd(V,e), where V is an integer variable and e is an integer expression. This indivisible operation yields the sum S=V+e as its value and replaces the contents of storage location V by this sum. Moreover, RepAdd must satisfy the serialization principle: Assume that V is a public variable (as it ordinarily will be) and many (perhaps very many) replace-add operations simultaneously address V. Then the effect is as if these operations occurred in some (unspecified) serial order, i.e. V receives the appropriate total increment and each operation yields the intermediate value of V corresponding to its position in this order[1]. The following example illustrates the semantics of replace-add: If V is a public variable, if PEi executes

ANSi <-- RepAdd(V,ei) ,

if PEj simultaneously executes

ANSj <-- RepAdd(V,ej) ,

and if V is not simultaneously updated by another PEk, then either

ANSi <-- V+ei
ANSj <-- V+ei+ej

or

ANSi <-- V+ei+ej
ANSj <-- V+ej

and, in either case, the value of V becomes V+ei+ej. The first possibility corresponds to the serialized order in which first PEi executes its replace-add and then PEj executes its replace-add; the second possibility corresponds to the opposite serialization. Suppose, to be still more specific, that V initially contained the value 10, and that ei=2 and ej=6. Then, after the simultaneous executions, V will contain 18 and either ANSi=12 and ANSj=18 or ANSi=18 and ANSj=16.

To further illustrate this semantic rule, we consider the execution of RepAdd(I,3-I). At first glance one might assume that this is simply another way of assigning 3 to I. However, if I is a public variable, it is possible for its value to change subsequent to evaluation of the expression 3-I but prior to the assignment to I that is triggered by the replace-add. Thus the value I returned by RepAdd(I,3-I) could in fact be different from 3.

It is also possible to have loads, stores, and replace-adds all concurrently directed at the same memory location. Once again the serialization principle demands that the effect is as though these operations occurred in some serial order. In particular, simultaneous loads from the same memory location may not yield identical results (since a simultaneous store or replace-add may intervene).

In section 12 we present a hardware design in which the replace-add operation requires essentially the same execution time as a load or store and in which simultaneous replace-adds updating the same variable are processed particularly efficiently.

2.3. Replace-Add as a Machine Instruction

The assembly level syntax of the replace-add operation in our (CDC-COMPASS-like) assembly language (Gottlieb [80]) is

REPAD Xi,Bj+K .

The effect of this hypothetical machine instruction is to replace both the contents C of register Xi and the

[1]These intermediate values result from executing prefixes of the serialized list of operations.

contents D of the effective memory location (determined by the address K and register Bj) by the sum C+D. That is, during one (indivisible) machine operation, the value in the memory location is added to the register and the sum is stored into both the memory location and the register. Note that these registers are local to each processor. Assume, for example, that three processors concurrently execute

REPAD X6,B0+10000 ,

where location 10000 is public and contains 10, and each processor's X6 register initially contains 3. Then the serialization principle guarantees that after execution of these three REPADs, location 10000 will contain 19 and the X6 registers in the three processors will contain, 13, 16, and 19. However, it is not specified which X6 register contains 13, which 16, and which 19.

3. Semaphores

Having reviewed the basic replace-add operation, we proceed to describe its role in implementing a variety of higher-level programming operations. We first present a replace-add based implementation of Dijkstra's [65] P(S) and V(S) operations (thus illustrating that replace-add obviates one important need for test-and-set), and then generalize this implementation to PVchunk operations PC(S,e) (resp. VC(S,e)) where S is incremented (resp. decremented) by e (see Vantilborgh and vanLamsweerde [72]). Subsequent sections show that our implementation of PC and VC permits more parallelism than traditional implementations.

Recall that the P and V operations are used to protect critical code sections by enforcing the following ''PV-property''. If many processors concurrently execute[2]

```
Procedure  PVTest
        Comment: Initially S=1.
        Loop    { P(S)
                critical section
                V(S) }
End Procedure   ,
```

if the critical section does not modify S, and if no PE ceases execution, then at any time T at most one processor is executing its critical section and there exists a time $t \geq T$ when exactly one processor is executing a critical section. (Note that this definition permits unfair scheduling.)

3.1. Implementing PV

In this section we present a PV implementation and in appendix A we prove that it satisfies the PV property. The P(S) operation first waits until the public variable S equals 1 and then executes RepAdd(S,-1). If the result is zero, the critical section may be entered. If the result is negative, some

[2] We use ''**{**'' and ''**}**'' for the tokens ''**Begin**'' and ''**End**'' respectively. However, our indentation convention obviates the need for these tokens.

other processor has control of the section and so P(S) ‘‘covers its tracks’’ and then tries again. The V(S) implementation consists simply of a replace-add incrementing S by 1. The following code is an appropriate implementation of these important primitives. (As will be explained below, various subtleties are involved.)

```
Procedure P(S)
    OK <-- False
    Repeat If  S-1 ≥ 0 Then
        If RepAdd(S,-1) ≥ 0  Then OK <-- true
        Else RepAdd(S,1)
    Until OK
End Procedure


Procedure V(S)
    RepAdd(S,1)
End Procedure
```

To emphasize a subtle point inherent in our implementation of P, consider the following very similar, but actually incorrect, implementation.

```
Procedure NaiveP(S)     Comment: Incorrect implementation of P
    OK <-- false
    Repeat If  RepAdd(S,-1) ≥ 0  Then OK <-- true
        Else RepAdd(S,1)
    Until OK
End Procedure
```

If one compares this simplified form with the correct original shown earlier, it may appear that we have merely removed a ‘‘redundant’’ test. However, the simplified code can in fact fail due to unacceptable race conditions. Suppose, for example, three PEs, A, B, and C, execute P(S) at the same time with S having its initial value of 1. If the serial order effected is equivalent to A executes first followed by B and C, then S is set to -2 and A enters the critical section. Suppose that A subsequently leaves the critical section, thus incrementing S to -1. The section should now be free to be entered by either B or C. The code above will allow this to occur as soon as S is incremented to +1 from its current value of -1. However, this may never happen, since the following endless scenario is possible: B increments S to 0 and then decrements S back to -1 before C executes its next instruction; thus B fails to enter the critical section. Then, while B is between instructions, C increments and immediately decrements S. B and C continue in this fashion indefinitely causing S to vary between 0 and -1, never reaching +1. Since every decrement occurs when S=0, the critical section is never entered and thus the PV-property is not satisfied.

This race condition, unlikely when just three PEs are involved, becomes steadily more probable as we increase the number of PEs trapped in the semaphore: The probability that some PE has executed the first replace-add but not the second rises with the number of PEs present.

Another plausible but not entirely satisfactory semaphore implementation is as follows: P(S) continually executes RepAdd(S,-1) until the result is 0; V(S) sets S to 1, representing an open gate; and S is initialized to 1. This scheme suffers from an acute danger of underflow if the wordsize is not large (e.g. if each PE in a 64K multiprocessor executed one RepAdd(S,-1) per microsecond, a 32 bit word would underflow within .1 seconds). Although larger wordsizes would ameliorate the difficulty, we do not consider this implementation further since it is not clear how to obtain the generalized semaphore that we discuss next.

3.2. Implementing PVchunk

In order to solve the readers/writers and other synchronization problems, it is convenient to define PVchunk operations where the increment e applied to the public variable S is not restricted to ±1. We write these operations as PC and VC and implement them using the same test-modify-retest paradigm seen above for P and V. The following code assumes that S has been initialized to some positive integer.

```
Procedure PC(S,e)
        OK <-- false
        Repeat If S-e ≥ 0 Then
                If RepAdd(S,-e) ≥ 0  Then OK <-- true
                Else RepAdd(S,e)
        Until OK
End Procedure
Procedure VC(S,e)
        RepAdd(S,e)
End Procedure
```

The PV operations can then be realized by defining P(S) to be PC(S,1), V(S) to be VC(S,1), and by initializing S to 1. A (slightly) more general mechanism, CP and CV (sometimes called a counting semaphore), which allows up to n processors to execute a ‘‘semi-critical section’’ simultaneously, is obtained by letting CP(S) and CV(S) be PC(S,1) and VC(S,1), respectively, and initializing S to n. Of course the standard ‘‘test and set’’ operation is adequate for implementing all these operations; however, it does not permit critical-section-free implementations of the higher level algorithms discussed in subsequent sections.

3.3. Remarks

It is worth noting that Dijkstra [72] considered the replace-add operation and examined the NaiveP procedure considered above, noting essentially the same race condition that we have discussed. Dijkstra concluded that the replace-add was a less appropriate coordination primitive than a simpler ‘‘swap’’ instruction. However, this conclusion becomes progressively less acceptable as the number of PEs grows larger.

As with other semaphore implementations, our PC/VC scheme suffers from possible starvation (also referred to as lockout or unfair scheduling). Since concurrent RepAdds are serialized in an arbitrary order, it is possible for PEj to be granted access to a critical section even though another PEi has been waiting for a longer period of time. In fact, as long as there are other PEs waiting, PEi may never be allowed to enter the critical section. However, the race condition described in section 3.1 cannot occur; no PE can wait for a permanently unoccupied critical section. A starvation-free semaphore based on the replace-add operation is presented in Rudolph [82].

4. The Test-Modify-Retest Paradigm

The previous section showed the need to test a semaphore before a decrement-and-test operation is applied. Since such test-decrement-retest (and corresponding test-increment-retest) sequences occur often, we define two procedures, each embodying one of these two basic sequences, which are used throughout the remainder of this paper.

```
Boolean Procedure TDR(S,Delta)
        TDR <-- false
        If S-Delta ≥ 0  Then
                If RepAdd(S,-Delta) ≥ 0 Then TDR <-- True
                Else RepAdd(S,Delta)
End Procedure
Boolean Procedure TIR(S,Delta,Bound)
        TIR <-- false
        If S+Delta ≤ Bound  Then
                If RepAdd(S,Delta) ≤ Bound Then TIR <-- true
                Else RepAdd(S,-Delta)
End Procedure
```

Using TDR the PC procedure of section 3.2 can be expressed simply as:

```
Procedure PC(S,e)
        Repeat Until TDR(S,e)
End Procedure
```

As noted above this formulation of PC suffers from the possibility of starvation, a difficulty affecting many loops using TDR as exit condition. However, we reemphasize that starvation, while undesirable, must be contrasted with the unacceptable race condition inherent in NaiveP: If one PE is starved, some other PE progresses. Moreover, as already noted, starvation-free algorithms do exist.

5. Readers and Writers

In preparation for the more complex problems to be considered below, we now use the PC and VC operations to solve the well known readers-writers problem, in which a group of ‘‘reader’’ processes and ‘‘writer’’ processes are to share the use of a resource. Many readers may use the resource simultaneously, but all other processes become blocked as soon as a single writer is active.

5.1. The Algorithm

The basic idea is to maintain a counter equal to n(1-w)-r, where n is (no less than) the maximum possible number of active readers in the system, and r and w equal the number of active readers and writers respectively.

The basic readers-writers problem has then the following simple solution (where S is initially n):

Procedure Reader
    PC(S,1)
    read-body
    VC(S,1)
End Procedure
Procedure Writer
    PC(S,n)
    write-body
    VC(S,n)
End Procedure

Note that, in the absence of writers, no serial code is executed by the implementation above. In contrast, standard ‘‘test and set’’ implementations use (very small) critical sections to protect the adjustment of their counters.

5.2. Priorities

The readers-writers solution given above allows a continuing stream of readers to lockout all writers and vice-versa. To prevent the former one can maintain another counter #W equal to the number of executing writer tasks and can then force potential readers to wait until no writer tasks are executing. The detailed code for this variant, which gives priority to writers, (and which we use in section 8) is as follows (initially, S=n and #W=0):

Procedure WPReader
    Repeat Until #W = 0
    PC(S,1)
    read-body
    VC(S,1)
End Procedure
Procedure WPWriter
    RepAdd(#W,1)
    PC(S,n)
    write-body
    VC(S,n)

RepAdd(#W,-1)
End Procedure

Naturally, readers can be given priority by proceeding symmetrically, i.e. by maintaining a counter equal to the number of executing readers. These algorithms are critical-section-free analogues of the algorithms presented by Courtois, Heymans, and Parnas [71]. A variant of this synchronization code in which no PE is ever starved is given in Rudolph [82]. Similar starvation-free variants also exist for many of the algorithms discussed below.

6. Management of Highly Parallel Queues

Although at first glance the important problem of queue management may appear to require use of at least a few inherently serial operations, we show in this section that a queue can be shared among processors without using any code that might create serial bottlenecks. The procedures to be shown next maintain the basic first-in first-out property of a queue, whose proper formulation in the assumed environment of large numbers of simultaneous insertions and deletions is as follows: If insertion of a data item p is completed before insertion of another data item q is started, then it must not be possible for a deletion yielding q to complete before a deletion yielding p has started.

Since queues are the central data structure for many algorithms, a concurrent queue access method can be an important tool for constructing parallel programs. When analyzing one of their parallel shortest path algorithms, Deo et al. [80] dramatize the need for this tool.

> ‘‘However, regardless of the number of processors used, we expect that algorithm PPDM has a constant upper bound on its speedup, because every processor demands private use of the Q.’’

6.1. The Algorithm

In the algorithm below we represent a queue of length Size by a public circular array Q[0:Size-1] with public variables I and D pointing to the locations of the items last inserted and deleted (these correspond to the rear and front of the queue respectively). Thus MOD(I+1,Size) and MOD(D+1,Size) yield the locations for the next insertion and deletion, respectively. Initially I=D=0 (corresponding to an empty queue).

We maintain two additional counters, #Ql and #Qu, which give lower and upper bounds respectively on the number of items in the queue and which never differ by more than the number of active insertions and deletions. Initially #Ql=#Qu=0, indicating no activity and an empty queue. The parameters QueueOverflow and QueueUnderflow, appearing in the code shown below, are flags denoting the exceptional conditions that occur when a processor attempts to insert into a full queue or delete from an empty queue. (Since a queue is considered full when #Qu ≥ Size and since deletions do not decrement #Qu until after they have removed their data, a full queue may have cells that could be used by another insertion.) The actions appropriate for the QueueOverflow and QueueUnderflow conditions are application dependent: One possibility is simply to retry an offending insert or delete; another possibility is to proceed to some other task. Note the former action can lead to starvation since one obtains a loop having exit condition TIR or TDR (cf. the end of section 4).

Code for a critical-section-free implementation of Insert and Delete is given below. The insert operation proceeds as follows: First a TIR is used to guarantee the existence of space for the insertion, and to increment the upper bound #Qu. If the TIR fails, a QueueOverflow occurs. If it succeeds, the expression Mod(RepAdd(I,1),Size) gives the appropriate location for the insertion, and the insert procedure waits its turn to overwrite this cell (this point is discussed below, see ‘‘cell contention’’). Finally, the lower bound #Ql is incremented. The delete operation is performed in a symmetrical fashion; the deletion of data can be viewed as the insertion of vacant space.

```
Procedure Insert(Data,Q,QueueOverflow)
        If TIR(#Qu,1,Size) Then {
                MyI <-- Mod(RepAdd(I,1),Size)
                Wait turn at MyI
                Q[MyI] <-- Data
                RepAdd(#Ql,1)
                QueueOverflow <-- False }
        Else    QueueOverflow <-- True
End Procedure
Procedure Delete(Data,Q,QueueUnderflow)
        If TDR(#Ql,1) Then {
                MyD <-- Mod(RepAdd(D,1),Size)
                Wait turn at MyD
                Data <-- Q[MyD]
                RepAdd(#Qu,-1)
                QueueUnderflow <-- False }
        Else QueueUnderflow <-- True
End Procedure
```

6.2. Cell Contention

Since we assume that PEs can execute at widely differing rates (due, for example, to memory contention, see section 12), it is possible for many active insert and delete operations to be assigned the same queue cell location. When the queue is nearly full or empty, conflicts involving one insert and one delete are reasonably likely (but a simple ‘‘cell-vacant’’ flag would be sufficient to resolve them). However, the circular array structure allows the (unlikely) possibility that many active insert and delete operations all attempt simultaneously to address the same cell. Four solutions to this conflict problem are presented in appendix B.

6.3. Avoiding Integer Overflows

Care is required to avoid potential overflows of the I and D counters caused by the combination of small word size, large numbers of processors, and high queue insertion rate: If each of 64K PEs inserts one item per millisecond, a 32 bit I counter overflows in 1 minute. Fortunately, we need only maintain the values of I and D modulo the queue size. One can simply ignore overflows if the result of an overflow is accurate modulo the machine wordsize if the queue size divides the wordsize. A less hardware-dependent approach is to insert the statement:

If I ≥ MaxInt-#PE **Then** RepAdd(I,-Size) ,

where MaxInt is the largest representable integer, immediately before the statement that increments I. Since many, even all, processors may execute this statement simultaneously, we require that

$$\text{MaxInt-\#PE-\#PE}*\text{Size} \geq \text{MinInt}$$

i.e. that

$$\text{MaxInt - MinInt} \geq \text{\#PE}*(\text{Size}+1) \quad .$$

Note that, if the range of representable integers is larger, we can optimize slightly by inserting

If I ≥ MaxInt-#PE **Then** RepAdd(I,-k∗Size)

instead, where k is some integer greater than one. A corresponding statement can be inserted to prevent D from overflowing.

7. A Remark on Experience using Parallel Synchronization Primitives

We first became aware of the potential importance of code permitting highly concurrent queue access during the development of a parallel program for radiation transport. These problems are completely parallelizable:[3] Since computations on separate particles are independent, any number of PEs can analyze particles asynchronously. The radiation transport program with which we were experimenting maintains a pool of particles; during processing, each PE deletes a particle from the pool, calculates, and inserts zero or more new particles back into the pool. A queue was used to represent the pool and, since the access routines were very short, we initially treated the queue as a serially reusable resource (i.e. critical sections were used). However, simulation (see Gottlieb [80]) of our programmed solution for this problem did not yield the expected linear speed-up: Addition of PEs beyond a critical number (depending upon the complexity of the physics calculations) did not decrease execution time by the expected amount, since serial queue access had become a significant bottleneck. But subsequent use of the highly concurrent queue routines shown above restored the originally expected linear speed-up. Figure 1 illustrates some of the results obtained.

---

[3] I.e. the speedup obtained by use of P PEs is $\Theta(P)$.

Figure 1. Serial vs. concurrent queues

8. An Operating System Scheduler

In order to define a queue-like data structure appropriate for the scheduler component of a highly parallel operating system, we need to enhance the queue mechanism described above to permit insertions of items tagged with priorities and multiplicities. Concerning priorities, we assume the ‘‘pleasant special case’’ (Knuth [73]) in which the set of priorities is finite and fixed in advance. A ‘‘queue with multiplicity’’ or ‘‘multiqueue’’ is a queue in which each item i has an associated multiplicity mi indicating the number of times i is to be deleted before it is removed from the queue, i.e. the pair (i,mi) represents mi consecutive entries of item i in a much longer (hypothetical) queue.

For expository purposes we begin by describing separately queues with priority and queues with multiplicity. In section 8.3 we combine these two concepts and show how the composite can form the core of an operating system scheduler.

8.1. Queues with Priority

The following highly concurrent implementation of a queue with priority maintains a set of queues, one for each priority, and supports insertions into any queue and ‘‘priority deletions’’ (i.e. deletion requests directed at the highest priority nonempty queue).

Suppose, for convenience, that the set of priorities is

$$\{0,1,...,\#P-1\},$$

where $\#P=2^s$ and 0 is the highest priority. To implement Q, the corresponding queue with priority, create #P queues Q[0], Q[1], ..., Q[#P-1] in the manner described in section 6, associating priority i with Q[i]. Then naive implementations of PrioInsert and PrioDelete are, respectively, to insert an item with priority i into Q[i] but to delete items by attempting to delete from Q[0], Q[1], ... until a nonempty queue is found. However, if #P is large and the high priority queues are frequently empty, this linear search is expensive and the following more complicated search (essentially a binary search) is preferable.

Establish a complete binary tree of depth D having the #P queues as leaves. Associate with each node in this tree a counter C recording the total number of items in the queues belonging to the subtree spanned by the node; in particular, the root contains the current size of Q. Whenever an item is inserted into Q[i], increment the counters in the nodes constituting the path from Q[i] to the root. Then a delete operation can find the highest priority nonempty queue by descending the tree using the counters as a guide. During descent the counters associated with the nodes traversed are decremented to record the imminent removal of an item.

The following code assumes that leftmost leaves have highest priority.

```
Procedure PrioInsert(X,Prio,Q,Overflow)
        Insert(X,Q[Prio],Overflow)
        If Not Overflow Then  {
                S <-- the prio-th leaf
                Loop    { RepAdd(C[S],1)
                        When S=Root Then Exit Loop
                        S <-- Parent(S) }}
End Procedure
Procedure PrioDelete(X,Q,Underflow)
        S <-- Root
        If TDR(C[S],1) Then {
                While S is not a leaf  {
                        If TDR(C[Left(S)],1) Then S <-- Left(S)
                        Else If TDR(C[Right(S)],1) Then S <-- Right(S) }
                Delete(X,Q[S],Underflow) }
        Else Underflow <-- True
End Procedure
```

Note that the delete operation invoked in PrioDelete never underflows since the counters guarantee the presence of an item (for a leaf S, C[S] serves as #Ql). However, this scheme does suffer from the possibility of starvation (although only in very unlikely situations).

8.2. Multiqueues

Next we turn our attention to queues with multiplicity (multiqueues). To implement such data structures we must be able to calculate deletion sites. For the queue implementation of section 6, deletion sites can be trivially calculated as one plus the number of previous deletions modulo the queue size. For multiqueues, the situation is more involved since many deletions can be directed at the same site. We propose two implementations for multiqueues.

8.2.1. A Simple Linear Implementation If items are typically inserted with large multiplicities, the following simple scheme is appropriate. Items augmented with their multiplicity counts are inserted into the multiqueue using essentially the algorithm given in section 6. A multiqueue deletion (‘‘m-deletion’’), after ascertaining that the multiqueue is nonempty, repeatedly applies TDRs until it finds that the head entry of the multiqueue has positive multiplicity. The last m-deletion for each item increments the front pointer by one (this is analogous to the action of Delete in section 6). Since each slot in the multiqueue can represent many items, #Ql and #Qu are replaced by #Qil, a lower bound on the number of items in the multiqueue, and #Qsu, an upper bound on the number of slots in use. We also associate with each slot J counters #RSD (resp. #RFD) representing the number of processes remaining to start (resp. finish) deleting items from slot J. In contrast with the situation described in section 6, concurrent deletes are all at one site and hence the insert pointer cannot pass the delete pointer. Thus, the issue of cell contention is simplified. However, we need to use writer priority readers-writers code (see section 5.2) when accessing the delete pointer in order to prevent the following senario: First two deletes concurrently obtain the same deletion site S in a full multiqueue; then one of the deletes obtains the sole item in this site, completes its deletion, and increments the delete pointer (thus making the multiqueue not full); then an insert begins, enters an item into site S, and completes (thus making the multiplicity of S positive); finally, the second delete checks the multiplicity of S for the first time, finds it positive, and deletes this new item from S instead of the older items in S+1.

In the code that follows the multiqueue is represented by the public circular array MQ[0:Size-1] and the multiplicity of an item to be inserted is denoted m. The body of the referenced WPReader routine is

```
MyD <-- D+1 mod Size
My#RSD <--- RepAdd (MQ.#RSD[MyD],-1)
```

and the body of WPWriter is

```
D <-- D+1 mod Size
RepAdd (#Qsu,-1)
```

The reader may check that the code remains correct if either (but not both) of the two statements in WPWriter are moved outside.

```
Procedure MInsert(Data,m,MQ,Overflow)
        If TIR(#Qsu,1,Size) Then  {
                MyI <-- RepAdd(I,1) mod Size
                MQ.data[MyI] <-- Data
```

```
        MQ.#RFD[MyI] <-- m
                MQ.#RSD[MyI] <-- m
                RepAdd(#Qil,m)
                Overflow <-- False  }
        Else Overflow <-- True
End Procedure
Procedure MDelete(Data,MQ,Underflow)
        If TDR(#Qil,1) Then {
                Repeat WPReader
                Until My#RSD ≥ 0
                Data <-- MQ.data[MyD]
                If RepAdd(MQ.#RFD[MyD],-1) = 0  Then WPWriter
                Underflow <-- False }
        Else Underflow <-- True
End Procedure
```

When the inserted items have small multiplicities, sequentially incrementing the delete pointer may create a significant serial bottleneck. Specifically, the time required for an m-deletion is proportional to the number of currently active deletion sites. In this case, the following more complicated scheme, which maintains an auxiliary binary tree T and thus requires time logarithmic in the size of the queue for both m-insertions and m-deletions (cf. ‘‘queues with priority’’ above), may be preferable.

8.2.2. A Logarithmic Implementation

We first describe the idea behind this logarithmic scheme and then discuss some of its finer points. The multiqueue is represented by the public circular array MQ[0:Size-1] and by a binary tree whose leaves correspond to the elements of MQ, with leftmost leaves corresponding to low values of the subscript. Each node of the tree contains a count of the number of items inserted in the cells corresponding to the leaves of its subtree. An m-insertion of an item i with multiplicity m first determines and reserves the next free queue cell MQ[j] (via a variant of the queue insertion algorithm of section 6). After the item and its multiplicity are placed into MQ[j], the m-insertion routine ascends the binary tree, beginning at the leaf corresponding to MQ[j] and ending at the root, and increments by m the counter in each node traversed. An m-deletion finds its deletion site MQ[j], corresponding to the leftmost nonzero count, by descending the tree using the counters as a guide. (Note that since multiplicities are allowed, several deletions may be directed to the same site.) During the descent, the counters in each node traversed are decremented by one to record the imminent deletion of an item. Each m-deletion returns a copy of the item in its deletion site and the last m-deletion at each site performs the standard queue deletion bookkeeping (c.f. section 6).

To handle queue wraparound properly, we maintain two trees and two circular arrays and let no m-deletion access one tree until all m-deletions have completed their actions on the other tree. That is, we impose ‘‘the one phase at a time’’ rule on deletions (c.f. appendix B.4). An m-insertion determines its assigned array (and corresponding tree) by using the value of I while an m-deletion determines (the root of) its tree by using WPReader (see section 5.2) with the read-body as follows:

```
MyRoot <-- Root
FoundRoot <-- TDR(C[MyRoot],1)
```

In addition, the last m-deletion addressed to one tree opens up the other tree for m-deletions by using WPWriter (see section 5.2) with write-body as follows:

```
Root <-- the other root
#Done <-- 0
```

Code for m-insertions and m-deletions, which utilizes this reader/writer algorithm, is as follows:

```
Procedure MInsert(Data,m,MQ,Overflow)
    If TIR(#Qu,1,Size) Then {
        MyI <-- RepAdd(I,1)
        MyRoot <-- Root of tree ( ⌊MyI/Size⌋ mod 2 )
        MyI <-- Mod(MyI,Size)
        J <-- MQ index corresponding to (MyRoot,MyI)
        MQ.data[J] <-- Data
        MQ.m[J] <-- m
        S <-- leaf corresponding to J
        While S is not a root {
            RepAdd(C[S],m)
            S <-- Parent(S) }
        RepAdd(C[MyRoot],m)
        RepAdd(#Ql,m)
        Overflow <-- False }
    Else Overflow <-- True
End Procedure
Procedure MDelete(Data,MQ,Underflow)
    If TDR(#Ql,1) Then {
        Repeat WPReader Until FoundRoot
        S <-- MyRoot
        While S is not a leaf {
            If TDR(C[left(S)],1) Then S <-- left(S)
            Else If TDR(C[right(S)],1) Then
                S <-- right(S) }
        J <-- MQ index corresponding to the leaf S
        Data <-- MQ.data[J]
        If RepAdd(MQ.m[J],-1) = 0 Then {
            RepAdd(#Qu,-1)
```

If RepAdd(#Done,1)=Size **Then** WPWriter **}**
Underflow <-- False **}**
Else Underflow <-- True
End Procedure

8.2.3. A Hybrid Implementation

A combination of the two multiqueue implementations given above is possible. Each m-deletion first accesses a short linear list and, if the list is empty, it then accesses the large binary search tree.

8.3. Scheduler Core

The algorithms just presented can readily be combined to implement a ‘‘multiqueue with priority’’ data structure: Use the queue with priority implementation, replacing the queues at the leaves of the priority tree P by multiqueues. That is, the leaves of P point to roots of multiqueue trees.

A major intended application for the data structures just described is the scheduling kernel of a highly parallel operating system. In this application the items inserted are task control blocks, the queue is called the task queue, and the following two primitives are supported:

RequestPE(N,P,CodeBlock)

whereby a request is made for N processes to execute a block of code at priority P, and

ReleasePE

whereby the PE invoking this primitive announces that it has completed its assigned task and is available for reassignment. The scheduler responds to the first primitive by inserting CodeBlock onto the task queue with priority P and multiplicity N. To implement the second primitive the scheduler deletes an entry from the task queue and transfers control to the corresponding CodeBlock.

As an example that can readily be organized using these primitives we note that, after some required initialization, the radiation transport problem mentioned in the last section can spawn multiple copies of a delete-physics-insert task, which can be executed in parallel. Another example arises in connection with relaxation techniques for 2-D PDE’s, which involve executing

$$x'(i,j) = (x(i-1,j)+x(i+1,j)+x(i,j-1)+x(i,j+1))/4$$

for $i,j = 1,2,...,n$. This can be parallelized by spawning n identical subtasks each of which determines a row index and calculates x' for the n grid points constituting this row, or even by spawning n^2 tasks each of which calculates x' at one point.

9. Highly Parallel Stack Operations

Next we show how to implement a stack for concurrent use by a large number of processors, in a manner which preserves the last-in first-out property of a stack even during concurrent access.

9.1. The Algorithm

We represent a stack of length Size by a public array S[0:Size-1] with a public variable Top indicating the current top of the stack. A push operation addresses stack location S[RepAdd(Top,1)], thereby incrementing Top. Similarly, a pop operation addresses stack location S[RepAdd(Top,-1)+1], thereby decrementing Top.

However, if simultaneous pushes and pops are interleaved they can all address the same stack location. To resolve the resulting conflicts, a queue into which pushes insert items and from which pops delete items is associated with each stack location. (At first we ignore space considerations in order to simplify the exposition.) These queues themselves can be managed by the critical-section-free algorithms of section 6.

As with queues, detection of stack overflow and underflow requires additional counters. We maintain #Su and #Sl as upper and lower bounds respectively on the number of elements in the stack. A TIR (resp. TDR) is applied to the upper (resp. lower) bound at the beginning of each push (resp. pop) operation. At the conclusion of the push (resp. pop) operation the lower (resp. upper) bound is incremented (resp. decremented).

Code for these stack management procedures follows.

```
Procedure Push(Data,S,StackOverflow)
        If TIR(#Su,1,Size) Then {
                Repeat  Insert(Data,S[RepAdd(Top,1)],Overflow)
                Until Not Overflow
                RepAdd(#Sl,1)
                StackOverflow <-- False }
        Else StackOverflow <-- True
End Procedure
Procedure Pop(Data,S,StackUnderflow)
        If TDR(#Sl,1) Then {
                Repeat  Delete(Data,S[RepAdd(Top,-1)+1],Underflow)
                Until Not Underflow
                RepAdd(#Su,-1)
                StackUnderflow <-- False }
        Else StackUnderflow <-- True
End Procedure
```

9.2. Storage Considerations

Since the queue associated with a stack location contains more than one element only when concurrent pops and pushes are interleaved, the queue size needed to maintain efficiency depends upon the pattern of stack activity. A queue size of #PE is always sufficient to avoid queue-overflows. A queue size reduced from this value still permits the code above to function correctly (albeit more slowly). It might appear that the delay caused by the occasional retry of a queue operation would violate the stack last-in first-out requirement. However, since two operations are considered concurrent if each begins before the other finishes, there does in fact exist some serial order of concurrent pushes and pops which does not violate the semantic definition of a stack. As usual these retries risk starvation.

10. Detecting Completion of Parallel Activity

Since the cessation of activity involving a shared resource will often be used to signal completion of a given task, it is important to be able to detect this event. To understand the ramifications of this remark, let us first consider the special problem of detecting a situation in which a shared queue is *and will remain* empty, namely when all the PEs are trying to delete from an empty queue. This is the natural termination condition for applications in which multiple PEs, each acting as both a producer and as a consumer, use a global queue to buffer data items which they pass among themselves. (The radiation transport problem considered above reveals the need for such a routine.)

10.1. The Algorithm

If the problem of detecting completion is ignored, the following code will typify such applications:

```
Loop    { If producer cycle Then  {
                produce data
                Repeat Insert(Data,Q,Overflow) Until Not Overflow  }
        Else { Comment: consumer cycle.
                Repeat Delete(Data,Q,Underflow) Until Not Underflow
                consume data  }}
```

However, the queue Underflow condition which this code generates is not sufficient to signify task completion since inserts may still occur even when the Underflow condition has been raised. Thus, to detect a state in which all PEs are trying to delete from an empty queue (denoted state T), we must modify the code shown above, which we do as follows. When a queue-underflow occurs, instead of retrying the delete, we increment a counter W which is then compared with #PE. If they are equal, state T has occurred. If not, the PE loops until either the queue becomes nonempty, in which case W is decremented and the deletion is retried; or until W equals #PE, in which case state T has occurred. The detailed code follows:

```
Comment Initially W = 0.
Loop    { If producer cycle Then  {
                produce data
                Repeat Insert(Q,Data,Overflow) Until Not Overflow  }
        Else { Comment: Consumer cycle.
                Repeat  Delete(Data,Q,QueueUnderflow)
                        If QueueUnderflow Then  {
                                RepAdd(W,1)
                                Repeat Until W = #PE Or #Ql > 0
```

```
                If W=#PE Then Comment state T.
                        Else RepAdd(W,-1)  }
        Until Not QueueUnderflow
        consume data  }}
```

Note that if P ≤ #PE tasks use this mechanism, they could execute the inner loop forever; in this case the code should be modified to compare W to P instead of to #PE.

10.2. A More General Problem

In addition to the termination condition discussed above, one may wish to detect the occurrence of a state T′ in which all PEs are attempting to insert into a full queue. In this connection we consider the more general situation in which any one of K ‘‘terminating’’ states are to be detected.

Suppose we are given K operations,

Op1, ..., OpK ,

each of which may succeed or fail (and be retried upon failure). We model this situation by having each PE execute

```
Loop    { calculate j, the operation index
        Repeat OPj Until success }  .
```

In order to detect the occurrence of any state Tj in which all PEs are unsuccessfully executing Opj, we assume that a failure predicate FPj (generalizing #Ql ≤ 0 above) is associated with each OPj, such that OPj fails if and only if it detects that FPj is true. The following code adapts the procedure used for detecting a permanently empty queue to the more general situation presently under discussion.

```
Loop    { calculate j
        Repeat  Opj(dataitems,Failure)
                If Failure Then {
                        RepAdd(Wj,1)
                        Repeat Until Wj = #PE Or Not FPj
                        If Wj = #PE Then Comment state Tj.
                        Else RepAdd(Wj,-1)  }
        Until Not Failure
```

11. Free Space Management

The queue operations described in section 6 may be applied to gain parallel access to the free space (avail) list used by the linked allocation scheme described in Knuth [73]. For expository purposes we assume that free space is allocated in fixed size blocks.

11.1. The Algorithm

We implement the avail list as a queue of pointers to free blocks and acquire (resp. return) blocks by deleting from (resp. inserting into) this queue of pointers. Unfortunately, for this implementation, a contiguous region of storage of size proportional to the maximum number of free blocks is required to maintain the queue of pointers.

11.2. Reducing the storage overhead

In many applications the number of free blocks needed vastly exceeds the maximum possible parallelism n (which is bounded by the product of the number of PEs and the maximum multiprogramming level). For these applications the size of contiguous storage needed can be reduced by implementing a parallel queue of size only n and having each entry provide exclusive access to one of the n conventional linked lists containing the free blocks. We note that each of these latter lists consists of essentially the same number of entries (see Rudolph [82] for details).

11.3. Applications

Once we can support concurrent access to the global avail list, it is possible to devise other highly parallel algorithms using linked allocation. When many PEs share access to a linked data structure L, it is often possible for them to perform insertions and deletions on L independently, provided that they lock list elements in the neighborhood of modification sites. If these sites are widely separated, the links in L can be updated in parallel. It is usually the case that insertions and deletions also require access to an avail list. Thus, the entire procedure can be performed without any serial bottlenecks by using the concurrent free-space management scheme described above. For a specific example consider concurrent insertions into an AVL tree. Ellis [80] shows how to parallelize the actual insertion process, and the techniques outlined in this section can be used to obtain the avail list elements needed to form new tree nodes.

12. Hardware Implementation

In this section we show how the replace-add operation used repeatedly in the preceding sections can be implemented using an ‘‘omega-network’’ enhanced so as to enable the network to process multiple replace-add operations in a highly parallel manner.

(See Benes [65], Lawrie [75] for a basic description of Omega-networks. The reader is also referred to Wu and Feng [80] for several topological equivalents to the omega-network each of which satisfies the fundamental property mentioned below (but with a different method for constructing the unique path). The results presented in this section apply equally well to these other networks. Moreover, readers familiar with recirculating networks may wish to adapt our techniques to those networks as well (see Siegel [76], Stone [71]).)

The first two parts of this section review the omega-network notion and explain how such networks can process multiple loads and stores efficiently. The third section details the enhancements needed to process replace-adds efficiently.

12.1. The Model

Suppose that $P = 2^D$ PEs are to communicate with a like number of memory modules (MMs); that both the PEs and the MMs are numbered using D-bit identifiers whose values range from 0 to P-1; and that the binary representation of each identifier x is denoted xD...x1. The (perfect) *shuffle* sigma, mapping the set {0,1,...,P-1} onto itself, is defined by

$$\mathrm{sigma}(xD...x1) = xD\text{-}1...x1xD \ ,$$

i.e. sigma performs a left rotation on the binary representation of x. Figure 2 illustrates the shuffle map for P = 8. This interconnection pattern, which dates back to Clos [53] and Benes [65], was further explored by Pease [67,68] and Stone [71].

We now describe the connection network to be used in our hardware design. An *omega-network* connecting P PEs to P MMs consists of D stages of two by two switches with adjacent stages connected via the inverse of the shuffle map[4]. As illustrated in figure 3 for P = 8, the PEs are directly connected to the first stage of switches and the last stage is connected via the inverse shuffle to the MMs. Each of the P/2 two by two switches constituting a single omega-network stage can transmit data in one of two modes, ‘‘straight’’, in which its top and bottom left terminals are connected to the top and bottom right terminals respectively, and ‘‘crossed’’, in which its top and bottom left terminals are connected to the

Figure 2. Logical schematic of perfect shuffle connections among eight processors.

[4]The network we consider is actually only ‘‘topologically equivalent’’ to an omega-network (see Wu and Feng [80]).

Figure 3. An 8-input omega-network.

bottom and top right terminals respectively (see figure 4). We show below that an omega-network provides a connection between any PE-MM pair.

We define a *memory cycle* to be the time required for a single PE, in the absence of any other communication traffic, to transmit a request to an MM and then receive a response. This cycle time equals the MM access time plus twice the network transmission time.

12.2. Implementing Loads and Stores

The manner in which an omega-network can be used to implement memory loads and stores is well-known (see Lawrie [75]). The existence of a (unique) path connecting each PE-MM pair is the key to an omega-network's use in connecting processors and memories. The path from PE(pD...p1) to MM(mD...m1) is constructed by setting the appropriate stage k switch to the straight position if pk = mk and to the crossed position if pk $\neq$ mk. Requests from PEs to MMs are transmitted along these paths and responses are transmitted along the reverse paths. Unfortunately, however, it is possible for two

Figure 4. The two possible states of an elementary omega-network switch.
(a) Switch in ''straight'' state. (b) Switch in ''crossed'' state.

concurrent requests to conflict, i.e. for the paths corresponding to these two requests to pass through the same switch but require different mode settings. For example, if PE0 references MM1 and PE1 references MM3 then a conflict arises at the stage 1 switch connected to PE0 and PE1. To satisfy the MM1 request this switch would have to be in the crossed position but to satisfy the MM3 request it would have to be in the straight position.

One way to resolve these conflicts is to ‘‘kill’’ one of the two conflicting requests and have it resubmitted by the PE. Despite the primitive nature of this scheme, proposed by Burroughs Corporation for their FMP [79], it exhibits good average case behavior: An easy analysis (see e.g. Schwartz [80b] and Valiant [80]) shows that, if the PEs randomly request unique MMs, approximately 1/4 are successful for P = 1024 and approximately 1/5 are successful for P = 64K. Moreover, the effective bandwidth of the connection network can be increased by duplexing (or quadruplexing) and putting 2 (or 4) copies of each data item on the network with 2 (or 4) different priorities. This raises the fraction of successful requests to the following approximate levels (see Kruskal and Snir [a]):

1024 PEs, duplexed switch	1/2 successful
1024 PEs, quadruplexed switch	3/4 successful
64K PEs, duplexed switch	1/3 successful
64K PEs, quadruplexed switch	5/9 successful .

Alternatively, we may resolve conflicts by enqueuing one of the two conflicting requests in the switch at which they conflict. Gottlieb and Kruskal [80] have shown that, for a network whose nodes have unlimited queue capacity, the queuing delay adds only about 50% to the average transmission time, again assuming that the PEs request distinct MMs. In the sequel we do not use this queuing technique: For expository purposes, we prefer the simpler Burroughs approach.

It is worth noting that some conflicts are ‘‘favorable’’: When concurrent loads and stores are directed at the same memory location and meet at a switch, they can be combined (and thereby satisfied) without introducing any delay using a scheme described below. Moreover, by determining the most favorable serial order for these simultaneous requests, an enhanced switch can combine them efficiently. The actions appropriate for each favorable conflict are as follows (some of these optimizations appear in the CHoPP design, see Klappholtz [81]).

(1) Load-Load: Transmit one of the two (identical) loads and return to each the value obtained from memory.

(2) Load-Store: Transmit the store and return its value to satisfy the load.

(3) Store-Store: Transmit either store and ignore the other.

Favorable conflicts reduce communication traffic and thereby increase the percentage of satisfied requests. Since combined requests can themselves be combined, any number of concurrent memory references to the same location can all be satisfied in one memory cycle (assuming the absence of conflicts with requests destined for other memory locations).

The switches can be further enhanced to increase performance. Schwartz [80] proposed combining adjacent two by two switches in each stage to form four by four switches. This reduces the number of switches per stage to P/4 and leads to the communication network shown in figure 5. Since in this scheme a switch is connected by two paths to each of two switches in the next stage, conflicts arise only when three or four inputs to a stage i switch require transmission to the same stage i+1 switch. It can be shown that this leads to improved throughput. Schwartz's analysis is extended by Kruskal and Snir [a] who also analyze Lawrie's [75] suggestion to combine switches in adjacent stages, thus halving the number of stages and slightly reducing the probability of conflicts.

12.3. Implementing Replace-Add

The replace-add operation can be realized by augmenting the MMs with adders and connecting them, via an omega-network, to the PEs: When a RepAdd(X,e) operation is transmitted through the network to the MM containing X, the value of X and the transmitted e are brought to the MM adder, and the sum is both stored in X and returned through the network to the requesting PE.

Since we expect that concurrent replace-add operations will frequently reference the same memory location, efficient performance in the case of favorable conflicts is very important. Fortunately, by including memory and an adder in each switch, the network can achieve for replace-adds the excellent performance described above for loads and stores. (Note that, although we shall continue to use the term

Figure 5. A generalized omega-network based on the use of 4-input switches.

‘‘switch’’ for the devices located at the nodes of the enhanced omega-network, these devices are functionally closer to microprocessors than to simple switches and thus may introduce nontrivial delays.)

When two replace-adds referencing the same public variable, say RepAdd(X,e) and RepAdd(X,f), conflict at a switch, we effect the serialization order ‘‘RepAdd(X,e) immediately followed by RepAdd(X,f)’’. This is done as follows: The switch forms the sum e+f, transmits the combined request RepAdd(X,e+f), and stores the value f in its local memory (see figure 6). When the value Y is returned to the switch (in response to RepAdd(X,e+f)), Y is returned to satisfy the incoming request RepAdd(X,f) and Y-f is returned to satisfy the incoming request RepAdd(X,e). If there was no other conflict, Y = X+e+f; thus the values returned are X+e and X+e+f and the memory location X receives this Y value X+e+f. If other RepAdd(X,g) are simultaneously processed, the combined requests are themselves combined and the associativity of addition guarantees that the procedure gives a result consistent with the serialization principle. In figure 7 we illustrate the case of four simultaneous requests combining in the last two stages of the network.

Figure 6. Treatment of simultaneous replace-add operations addressing the same memory location.

Figure 7. Four replace-add operations combining while traversing the last two network stages.

In summary, favorable replace-add conflicts are processed as follows:

(1) RepAdd-RepAdd. As described above, a combined request is transmitted and the result used to satisfy both replace-adds.

(2) RepAdd-Load. Treat Load(X) as RepAdd(X,0).

(3) RepAdd(X,e)-Store(X,f). Transmit Store(X,e+f) and satisfy the replace-add by returning e+f.

As seen in our discussion of loads and stores, the scheme above reduces communications traffic and exhibits good average case performance.

The advantages of using four by four switches in the basic omega-network are also applicable to the enhanced network just described. A detailed analysis and hardware design of this enhanced network appears in Gottlieb et al. [81].

13. Summary

Since the relative cost of serial bottlenecks rises linearly with the number of PEs present, elimination of such bottlenecks will become steadily more important in future parallel processors. By exhibiting bottleneck-free implementations for several important operating system primitives, by noting that replace-add can also be used to define efficient parallel implementations of scientific application codes, and by presenting an efficient hardware realization of the replace-add operation, we hope to have shown that this operation is an appropriate synchronization tool for ultra-large scale parallel processors. We note that our replace-add implementation avoids the hardware bottleneck usually associated with concurrent access to a single memory location.

We believe that future parallel processors, utilizing something close to the hardware design presented in section 12, can realize the replace-add in very little more than the time required for a public memory reference. Since it is expected that on-chip delay times will typically be less than the chip-to-chip transmission times, the network overhead imposed by supporting the replace-add operation will not degrade network transmission time significantly. We note that the ‘‘ultracomputer’’ group at NYU is developing a preliminary design for a prototype machine and operating system incorporating the ideas presented above (see Gottlieb et al. [81]).

This paper has ignored the important issue of fault tolerance; the algorithms presented assume that the hardware functions correctly. We hope to describe more robust algorithms in future reports; it appears that such enhancements need degrade performance by only a (relatively small) constant factor.

14. Acknowledgement The authors thank Clyde Kruskal and Jack Schwartz for technical contributions; Kevin McAuliffe, Marc Snir, and the three anonymous referees for reading preliminary versions of this paper carefully; Jim Wilson for pointing out the Dijkstra [72] paper; and the entire NYU Ultracomputer group for many helpful discussions.

A. Proof of the PV-property

This appendix presents a proof of the following result, which was first stated in section 3.

Theorem. Replace-add implementation of PV satisfies the PV-property.

Proof. Assume each PE executes PVTest with only a finite delay between instructions. Expanding the invocations of P and V, we obtain the following code for each PE, where TDR, Temp1, and Temp2 are local variables.

```
         Comment: Initially S=1.
 1       Loop {
 2             Repeat TDR <-- False
 3                    Temp1 <-- S-1
 4                    If  Temp1≥0 Then {
 5                          Temp2 <-- RepAdd(S,-1)
 6                          If  Temp2≥0 Then
 7                                TDR <-- True
 8                          Else RepAdd(S,1) }
 9             Until TDR
10             critical section
11             RepAdd(S,1)  }
```

Continuing with our proof, we make the following

Claim. One may assume, without loss of generality, that during each time step, exactly one PE executes exactly one line of PVTest.

Proof of Claim. By the serialization principle we may assume that during each time step exactly one PE executes exactly one machine instruction. That is, we can assume a serial execution order I1,I2,... where each Ij represents execution of one machine instruction by one PE; of course the instructions executed by each PE are in the sequence determined by the PVTest code. Note that we can interchange any two consecutive instructions Ij and I(j+1) executed by two different PEs provided that at most one of them references a public variable. Thus, since each line of PVtest contains at most one reference to S, there exists a sequence of interchanges that yields an execution order in which each line of PVtest is executed indivisibly. This proves our claim.

To describe the state of a PE at time T, we introduce terminology for specifying each PEs location counter and the value of the shared variable S. For $1 \le j \le 11$, PEi is said to be *at j* if the next instruction to be executed by PEi is line j of the program above. We write Li(T) = j to indicate that at time T, PEi is at j. For j = 4, 6, and 9, the three decision points of the program, we distinguish two subcases: PEi is *at j+* (resp. *at j-*) if PEi is at j and the condition to be tested in line j is true (resp. false). The value of S at time T is denoted S(T) and equals 1 + SUM ci(T), where 1 is the initial value of S and ci(T) is the cumulative contribution to S caused by actions of PEi completed before time T. This latter quantity equals

$$ni(T,8) + ni(T,11) - ni(T,5) \quad ,$$

where ni(T,j) is the number of executions of line j by PEi completed before time T.

A simple analysis of PVTest as a sequential program yields

Proposition 1. For all times T and PE indices i, ci(T) is 0 or -1. Specifically, ci(T) = -1 if and only if Li(T) = 6, 7, 8, 9+, 10, or 11.

Corollary 1. At any time T, $-\#PE < S(T) \le 1$.

We now prove the easy half of the theorem, namely that mutual exclusion is guaranteed. We call a PE *critical at time T* if Li(T) = 6+, 7, 9+, 10, or 11 and define N(T) to be the number of such PEs.

Proposition 2. At any time T, $N(T) \le 1$.

Proof. If not, there exists a time t0 such that N(t0) = 1 and N(t0+1) = 2. Thus, for some processor PEi, Li(t0) = 5 and Li(t0+1) = 6+. But, by proposition 1, any critical PE contributes -1 to S and thus, S(t0) < 1. This contradicts the statement that PEi makes a transition from 5 to 6+ at time t0.

Corollary 2. At any time T, at most one PE can be executing its critical section.

To complete the proof of our theorem, we must show that some time after any reachable state is established, some PEi will enter the critical section. The key to this is to verify:

Lemma 1. For any time T there exists a time $T' \ge T$ such that $S(T') = 1$.

Proof. Suppose S(t) < 1 for all $t \ge T$. Then after time T, no PE can make the transition from state 3 to state 4+ or from state 5 to state 6+. Therefore the flowgraph for PVTest becomes

```
                                  ______________________
                                  |                     |
6+ --> 7 --> 9+ --> 10 --> 11 --> 2 --> 3 --> 4 --> 9 --
                                  |                 |
                                  1                 |
                                                    |
                               4+ --> 5 --> 6- --> 8  .
```

Hence, there exists $T' \ge T$ such that at time T' all PEs are at 2, 3, 4-, or 9- and thus, by proposition 1, $ci(T') = 0$ for all i. But this implies that $S(T') = 1$ as desired.

Finally, if T' is as in the lemma it is easy to see that after time T' the first PE to execute line 5 becomes critical, and then enters its critical section.

In this appendix we discuss the cell contention issues raised in section 6.2. Our first two solutions use semaphores to achieve exclusive access to each cell, the third solution uses an ‘‘activity counter’’, and the final solution imposes a ‘‘one phase at a time’’ discipline.

A.1. Solution 1

Cell vacant flags, one per cell, can be used to resolve cell contention: Insertions proceed only when the flag is set ‘‘vacant’’ and deletions proceed only when the flag is set ‘‘full’’. These flags must be protected by semaphores to avoid their being concurrently updated by multiple insertions and deletions.

Initially the flags are set to ‘‘vacant’’ and the semaphores are set to 1 (representing ‘‘open’’). Then the code to place data into cell Q[MyI] becomes

P(InsertSem[MyI])
Repeat Until CellVacant[MyI]
Q[MyI] <-- Data
CellVacant[MyI] <-- False
V(InsertSem[MyI])

and the code to extract data from cell Q[MyD] becomes

P(DeleteSem[MyD])
Repeat Until Not CellVacant[MyD]
Data <-- Q[MyD]
CellVacant[MyD] <-- True
V(DeleteSem[MyD]) .

A.2. Solution 2

In this refinement of the first solution we eliminate the cell vacant flag: We use an initially open insert semaphore and an initially closed (i.e. zero valued) delete semaphore to enforce both mutual exclusion and the alternation of inserts and deletes at the cell level. The completion of an insert at Q[i] resets the associated delete semaphore and the completion of a delete resets the corresponding insert semaphore. The code to place data in cell Q[MyI] is then

P(InsertSem[MyI])
Q[MyI] <-- Data
V(DeleteSem[MyI])

and the code to extract data from cell Q[MyD] is

P(DeleteSem[MyD])
Data <-- Q[MyD]
V(InsertSem[MyD]) .

A.3. Solution 3

Instead of using semaphores, this solution maintains in each cell an activity counter C equal to the number of completed operations (i.e. inserts and deletes) on the cell. Inserts wait until C becomes $2\lfloor I/Size \rfloor$, deletes wait until C becomes $2\lfloor D/Size \rfloor+1$, and both inserts and deletes increment C when completed. In the algorithm presented in section 6 we have calculated the local value MyI as RepAdd(I,1) modulo Size. Here we require the corresponding quotient as well. Thus, for Insert, we replace

MyI <-- Mod(RepAdd(I,1),Size)

by

```
MyIRaw <-- RepAdd(I,1)
MyI     <-- Mod(MyIRaw,Size)
MyPhase<-- MyIRaw Div Size
```

and we make the corresponding change for delete. Then the code to place an item into cell Q[MyI] becomes

```
Repeat Until C[MyI] = 2*MyPhase
Q[MyI] <-- Data
RepAdd(C[MyI],1)
```

and the code to extract data from cell q[MyD] becomes

```
Repeat Until C[MyD] = 2*MyPhase+1
Data <-- Q[MyD]
RepAdd(C[MyD],1)
```

In regard to the issue of integer overflows (cf. section 6.3), we note that MyPhase and the C counters need only be accurate modulo #PE and thus I and D need only be maintained modulo #PE times Size.

A.4. Solution 4

A final possibility is to group insertions and deletions into ‘‘phases’’ by their insertion or deletion numbers, each phase consisting of Size successive insertions and deletions, and then to delay phase p deletes until all phase p-1 deletes have completed. That is, deletes numbered $I=p*Size+r$ are delayed until all deletes numbered $I'=(p-1)Size+r'$ have completed, where $0 \le r,r' \le Size$. This implies that phase p inserts must wait for all phase p-2 inserts to finish. Since phase p inserts and phase p-1 inserts can be performed simultaneously, two cells must be used to represent each Q[I]. Insert-Delete contention is resolved using a cell vacant flag. This ‘‘one phase at a time’’ discipline, overly restrictive for the problems addressed in section 6, is used in section 8 to implement ‘‘multiqueues’’, where code for it can be found.

References

V. E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, NY, 1965.

James E. Burns, Michael J. Fischer, Paul Jackson, Nancy A. Lynch, Gary L. Peterson, ''Shared Data Requirements for Implementations of Mutual Exclusion Using a Test-and-Set Primitive'', *Proc. 1978 Intern. Conf. on Parallel Processing*, pp. 79-87.

Burroughs Corp., *Numerical Aerodynamic Simulation Facility Feasibility Study*, NAS2-9897, March 1979.

Charles Clos, ''A Study of Nonblocking Switching Networks'', *Bell Sys. Tech. J.* **32** 1953, pp. 406-424.

P. J. Courtois, F. Heymans, and D. L. Parnas, ''Concurrent Control With 'Readers' and 'Writers''', *CACM* **14** 1971, pp. 667-778.

Narsingh Deo, C.Y. Pang, and R.E. Lord, ''Two Parallel Algorithms for Shortest Path Problems'', *Proc. 1980 Intern. Conf. on Parallel Processing*, pp. 244-253.

E. W. Dijkstra, ''Solution of a Problem in Concurrent Programming Control'', *CACM* **8** 1965, p. 569.

E. W. Dijkstra, ''Hierarchical Orderings of Sequential Processes'' in *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrot Editors, Academic Press, NY, 1972.

E. W. Dijkstra, ''Self-Stabilizing Systems in Spite of Distributed Control'', *CACM* **17** 1974, pp. 643-644.

Danny Dolev, ''A Comparative Study of Synchronization by Parallel Control Systems'', Ph.D. Thesis, Weizmann Institute of Science, Rehovot, Israel, 1979.

E. Draughon, R. Grishman, J. Schwartz, and A. Stein, ''Programming Considerations for Parallel Computers'', Courant Institute, NYU, IMM 362, Nov. 1967

M. A. Eisenberg and M. R. McGuire, ''Further Comments on Dijkstra's Concurrent Programming Control Problem'', *CACM* **15** 1972, p. 999.

Carla Schlatter Ellis, ''Concurrent Search and Insertion in AVL Trees'', *IEEE Trans.* **C-29** 1980, pp. 811-817.

Allan Gottlieb, ''Washcloth - The Logical Successor to Soapsuds'', Ultracomputer Note #12, Courant Institute, NYU, 1980.

Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir, ''The NYU Ultracomputer -- Designing an MIMD Shared Memory Paralle Computer'', *IEEE Trans*. **C-32** 1983, pp. 175-189.

Allan Gottlieb and Clyde Kruskal, ''A Data Motion Algorithm'', Ultracomputer Note #7, Courant Institute, NYU, 1980.

Peter B. Henderson and Yechezkel Zalcstein, ''Characterization of the Synchronization Languages for PV Systems'', *Proc. 1978 Intern. Conf. on Parallel Processing*.

George Holober and Lawrence Snyder, ‘‘Scheduling Parallel Processes Without a Common Scheduler’’, *Proc. 1979 Intern. Conf. on Parallel Processing*, pp. 186-195.

Howard P. Katseff, ‘‘A New Solution to the Critical Section Problem’’, *Proc. of the 10th Annual ACM Symp. on Theory of Comp.*, 1978, pp. 86-88.

David Klappholz, private communication, 1981.

Donald E. Knuth, ‘‘Additional Comments on a Problem in Concurrent Programming Control’’, *CACM* **9** 1966, p. 321.

Donald E. Knuth, *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, Addison-Wesley, Reading MA, 1973 (page 153).

Clyde P. Kruskal and Marc Snir, ‘‘Analysis of Omega-type Networks for Parallel Processing’’, in preparation [a].

Leslie Lamport, ‘‘A New Solution of Dijkstra’s Concurrent Programming Problem’’, *CACM* **17** 1974, pp. 453-455.

Duncan Lawrie, ‘‘Access and Alignment of Data in an Array Processor’’, *IEEE Trans.* **C-24** 1975, pp. 1145-1155.

R. J. Lipton, ‘‘Limitations of Synchronization Primitives with Conditional Branching and Global Variables’’, *Proc. of the 6th Annual ACM Symp. on Theory of Comp.*, 1974, pp. 230-241.

R.J. Lipton, L. Snyder, and Y. Zalcstein, ‘‘Evaluation Criteria for Process Synchronization’’, *Proc. IEEE Sagamore Conf. on Parallel Processing*, 1975, pp. 245-250.

Marshall C. Pease, ‘‘Matrix Inversion using Parallel Processing’’, *JACM* **14** 1967, pp. 757-764.

M. C. Pease, ‘‘An Adaptation of the Fast Fourier Transform for Parallel Processing’’, *JACM* **15** 1968, pp. 252-264.

Gary Peterson and Michael J. Fischer, ‘‘Economical Solutions for the Critical Section Problem in a Distributed System’’, *Proc. of the 9th Annual ACM Symp. on Theory of Comp.*, 1977, pp. 91-97.

Ronald Rivest and Vaughan R. Pratt, ‘‘The Mutual Exclusion Problem for Unreliable Processors: Preliminary Report’’, *Proc. of the 17th Annual Symp. on Foundations of Comp. Sci.*, 1976, pp. 1-8.

Larry Rudolph, ‘‘Software Structures for Ultraparallel Computing’’, Thesis, NYU, 1982.

J. T. Schwartz, ‘‘Ultracomputers’’, *ACM TOPLAS* **2** 1980[a], pp. 484-521.

J. T. Schwartz, ‘‘The Burroughs FMP Machine’’, Ultracomputer Note #5, Courant Institute, NYU, 1980[b].

Howard J. Siegel, ‘‘Single Instruction - Multiple Data Stream Machine Interconnection Design’’, *Proc. 1976 Intern. Conf. on Parallel Processing*, pp. 272-280.

Harold S. Stone, ‘‘Parallel Processing with the Perfect Shuffle’’, *IEEE Trans.* **C-20** 1971, pp. 153-161.

Herbert Sullivan, Theodore Bashkow, and David Klappholz, ‘‘A Large Scale Homogeneous, Fully Distributed Parallel Machine’’, *Proc. of the 4th Annual Symp. on Comp. Arch.*, 1977, pp. 105-125.

H. Vantilborgh and A. vanLamsweerde, ‘‘On an Extension of Dijkstra’s Semaphore Primitives’’, *Inf. Proc. Let.* **1** 1972, pp. 181-186.

L. G. Valiant, ‘‘Experiments with a Parallel Communications Scheme’’, presented at the *18th Allerton Conf. on Communication, Control, and Computing*, 1980.

Chuan-lin Wu and Tse-yun Feng, ‘‘On a Class of Multistage Interconnection Networks’’, *IEEE Trans.* **C-29** 1980, pp. 694-702.

Table of Contents