Scalable Implementation of Synchronization Primitives on Broadcast Rings*

Martin H. Davis, Jr. *Umakishore Ramachandran*

GIT-CC-93/07

January 1993

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
{davism,rama}@cc.gatech.edu

Abstract

Synchronization is an important aspect of parallel program design. By definition synchronization is an aspect of a program where multiple processors participate. Thus it is important to design and implement hardware primitives that scale well with the size of the parallel machine, both in terms of space and time requirements. The focus of this research is to propose implementation for some well-known synchronization primitives in a broadcast ring network. The key aspects of the implementation are to make local decisions to determine the outcome of the synchronization operations; and to keep the space overhead per node constant independent of the number of processors participating in such operations. It is also shown that the implementation incurs exactly the minimum amount of communication to perform the synchronization operations.

Keywords: scalable synchronization primitives; broadcast ring interconnect; shared memory multiprocessor; distributed queues; local decision algorithms.

*This work has been funded in part by NSF PYI Award MIP-9058430.

1 Introduction

The work in this paper focuses on how shared memory style synchronization commands can be efficiently implemented. In order to make our solution general, we have made the implementation of these commands orthogonal to the physical implementation of the shared memory abstraction. In contemplating this problem, we have been interested in how the interconnection network can be structured so as to provide more than just communication services, i.e., how can the interconnection network's structure provide support for useful parallel programming commands? From this consideration we have developed interesting "local decision" techniques for implementing locks, barriers, and F&OP synchronization primitives.

In Section 2 we describe these implementation techniques. Our implementation incurs exactly the minimum amount of communication to perform the synchronization, by making local decisions for determining the outcome of the synchronization operations. Further, the implementation requires a constant amount of storage space per node independent of the number of processors participating in a synchronization operation. We present some discussion and analysis regarding these issues in Section 3. Finally, we conclude in Section 4 with questions for future work.

2 Synchronization Primitives

There are software algorithms for barriers and spin-lock that minimize the amount of network messages and storage space [4] for implementing such synchronization operations. However, such algorithms do not reduce the inherent latency for synchronization operations. This latency is two-fold: (1) the potential wait times at synchronization points due to simultaneous access from parallel processors; and (2) the intrinsic overhead for implementing the synchronization. In this section we describe our proposed hardware implementation for some well-known synchronization primitives that reduces this latency.

Our implementation techniques for the three synchronization primitives are all *local decision monitoring* based. That is, each node maintains its own *local* copy of a data structure, which represents the current state of a particular instance of the synchronization primitive, by *monitoring* all the nodes' commands which affect that particular instance.

The data structure contains enough information that each node independently *decides* how it is affected by the current state.

The only architectural requirement for our local decision scheme is that the nodes transmit their synchronization commands via what we call a *broadcast ring*. The broadcast ring has three properties which are illustrated in Fig. 1. First, the nodes are attached (at least logically if not physically) to the medium as a ring. Thus, a node's message is transmitted unidirectionally to all its downstream neighbors, and a node always hears the messages from all its upstream neighbors. Second, every node's message is broadcast to every other node, i.e., each node is connected to the broadcast ring such that it hears all messages (including its own). This connection is equivalent to every node having a connection so that every node (including itself) is upstream of the connection. Third, each node hears the broadcast messages in the same order as every other node; this property is expressed as a "train" of messages.

In the following subsections we describe these local decision monitoring algorithms for locks, barriers, and F&OP.

2.1 Locks

A *lock* is a mechanism by which a task can access a set of shared data associated with the lock in a controlled and regulated fashion. There are two different modes in which a lock can be requested: *read* (or shared, denoted as *R/S*) and *write* (or exclusive, denoted as *W/E*). By definition, use of the R/S mode guarantees to the task that the data cannot change for the duration of the lock. Similarly, by definition, use of the W/E mode guarantees to the task that no other task can access the data, much less change the data.

How requests for a particular lock are serviced can be understood by the concept of *peer groups*, as illustrated in Fig. 2. A peer group of lock requests is a set of consecutive and compatible requests for a particular lock. Thus, a W/E request forms a peer group of size one. R/S requests made without an intervening W/E request form a peer group of arbitrary size. Although peer groups can be serviced out of order, some rules must exist to prevent starvation of requests. From a performance standpoint, if a R/S peer group is the currently serviced peer group and no other peer groups are pending, then another R/S request joins the current R/S peer group and is granted the lock immediately.

Fig. 3 shows the data structure needed for each lock (note the various fields are grouped into four parts). Remember that this data structure is replicated at each node and modified independently by each node. The first group (`LOCK-ID` and `my`) contains the fields to identify the lock and define the peer group into which the lock request is placed. The second group (`prior`) of fields defines the peer group which is to be serviced immediately before the lock request's own peer group. The third group (`most-recent`) of fields defines the most recent peer group of lock requests, and the fourth group (`next-recent`) defines the peer group *just before* the most recent peer group. These four groups of fields, when taken as a whole over all the nodes, implement a distributed queue of lock requests. This queue is replicated at each node; note, however, that each node stores only the portion of the queue that is necessary to its knowing wherein the queue its own request falls and when its request should be granted.

To maintain its portion of the distributed queue, each node follows specific monitoring algorithms (shown in Appendix A) when observing all nodes' lock requests. Recall that the broadcast ring's properties guarantees that each node sees all the lock command traffic both preceding and succeeding its own command and that each node sees the same sequence of lock commands as every other node. The algorithms can be summarized as follows (the details can be found in [1]). The first algorithm is for when a node sees a lock request from any node except itself: the node updates the `most-recent` and `next-recent` groups. The second algorithm is for when a node sees its own lock request command: the node updates the `my` fields, the `prior`, `most-recent`, and `next-recent` groups, and determines whether it can immediately acquire the lock. The third and fourth algorithms are needed for when the node sees a R/S or W/E unlock command respectively: the `most-recent` and `next-recent` groups are always updated appropriately, and the node's own outstanding lock request (if any) is granted if appropriate. Thus, this implementation provides the logical abstraction of a nearly FCFS queue for lock requests. It is "nearly" FCFS because of the broadcast ring's possible re-ordering of nodes' requests within one ring cycle. Although the broadcast ring's unidirectional property compromises true FCFS service, the same property also guarantees forward progress and no starvation of the lock requests.

2.2 Barriers

The *barrier* is a rendezvous point for some set of tasks. The barrier's semantics are that after initialization of the barrier, the participating tasks are free to reach the barrier at their own pace. A task is in the *arrival* phase when it reaches the barrier. *Barrier completion* occurs when all participating tasks have arrived at the barrier. A task must wait at the barrier until it becomes *aware of* barrier completion, at which time it can proceed past the barrier.

Fig. 4 depicts the simple data structure which captures the state of a barrier. Recall that there is one such data structure per barrier, and this data structure is replicated at each node. The `ID` field uniquely identifies the barrier. The `participation` field stores how many nodes are participating in the barrier, and the `counter` field tracks how many nodes have reached the barrier. By maintaining the state of the barrier, each node can deduce when barrier completion occurs.

To maintain the state of the barrier, each node follows a simple monitoring algorithm (shown in Appendix B) when observing all nodes' barrier commands. If the observed barrier command is for barrier initialization, then the `participation` and `counter` fields are set to the number of participating nodes and zero respectively. If the observed barrier command is that a node has reached the barrier, then the `counter` field is incremented. When a node itself reaches the barrier, it stalls at the barrier by watching the `counter` field and waiting for it to equal the `participation` field. When the two fields are equal, the node realizes that barrier completion has occurred. Note that since each node locally ascertains when barrier completion has occurred, there is no need for an explicit command to notify nodes of barrier completion.

If the data structure continues to represent the same logical barrier, then reuse of the barrier hardware is a simple matter of each node resetting its copy of the `counter` field. If the data structure is to represent a different logical barrier, then an explicit barrier initialization command is required to reset the `participation` and `counter` fields.

2.3 Fetch-&-OP

The F&OP (Fetch-&-OP) primitive is a generalization of the Fetch-&-ADD primitive introduced by Gottlieb [2]. Give that the `OP` represents any associative, commutative operator,

the semantics is that when a node issues `F&OP(V,e)`, the command returns the *old* value of `V` to the node and atomically replaces `V` with `V OP e`. An attractive property of the F&OP command is its potential combining capability. This capability can be expressed by saying that the F&OP command must satisfy the serialization principle: if `V` is a shared variable and many nodes issue `F&OP(V,e)` to the same target `V` simultaneously, then the effect of the many parallel F&OP commands is exactly what it would be if they had occurred in some (unspecified) serial order. That is, the final value of `V` due to the parallel `F&OP(V,e)` commands is the result of applying all the operators `OP` and operands `e` to the original `V`, and each node receives an intermediate value of `V` depending upon where its own particular `F&OP(V,e)` command happens to fall in the arbitrary serial order.

The data structure associated with each target `V` is shown in Fig. 5. The `V-current` field stores the current value of `V`, and the `V-mine` field stores the value of `V` which the node obtains from having issued the `F&OP(V,e)` command. Before any `F&OP(V,e)` command for a particular target `V` is issued by a node, the target `V` must be initialized, i.e., the `V-current` field is set to the initial value of `V`. From this data structure, each node determines the value of `V` it should receive from its `F&OP(V,e)` command and updates `V`.

To update its data structure, each node follows a simple monitoring algorithm (shown in Appendix C) when observing all nodes' F&OP commands. When the node observes its own command, it sets `V-mine` equal to `V-current` (and returns that value to the CPU); for any node's command, the node updates `V-current`. Thus, each node is provided the logical abstraction that its F&OP command is placed somewhere into a serial ordering of all the nodes' F&OP commands.

3 Analysis and Discussion

3.1 General observations

There are several general observations to be made about these implementations. First, each implementation is decentralized. Each node independently maintains its copy of the appropriate data structure by observing the synchronization command traffic. Specifically, for locks, each node observes the stream of lock commands including its own. By observing this stream of commands, the node determines in what peer group its own command falls;

from the stream, the node also deduces when the lock is released by the previous holder(s). Thus, no central entity takes responsibility for queuing lock requests and granting them. For barriers, each node observes the stream of barrier arrivals including its own. By observing this stream, each node independently counts how many nodes have arrived at the barrier; from this information, each node independently ascertains when barrier completion has occurred. Thus, since no central entity computes barrier completion, no central entity has responsibility for notifying others of barrier completion. For the F&OP command, each node also observes the stream of commands including its own. By observing this stream, each node determines its own command's place which, in turns, allows the node to compute the value of its F&OP command. Since each node listens to all the F&OP commands, each node also maintains the current value of the F&OP targets, thus avoiding the need for a central entity to maintain the value. The key to this decentralized scheme's working is that each node hears all nodes' messages in the same order as every other node.

Recall that we are interested in how the interconnection network's structure provides support for the implementations. Our second observation concerns how the broadcast ring's property of ordering the messages plays a role in the implementations. In the case of locks, the ordering automatically forms the service order of peer groups; thus, the interconnection network takes the responsibility for arbitrating among the queue of lock requests. For barriers, the ordering plays no role. For the F&OP command, the ordering automatically creates the (arbitrary) serial ordering needed for each node to calculate locally the value of its `F&OP(V,e)` command.

The third observation concerns the space requirements. For the lock design, note that no matter how large the distributed queue of lock requests becomes, each node only needs to store four peer groups of information (per distinct lock). For the barrier, each node stores only one group per barrier no matter how many nodes participate in the barrier. For the F&OP, each node also stores only one group per target `V` no matter how many nodes issue F&OP commands to that target. Thus, for each synchronization primitive implementation, the data structure storage requirements are constant per node per instance of the primitive no matter how many nodes participate in operations on the primitive.

The fourth observation regards the number of messages generated for each primitive and how that number compares with the minimum number needed. For either a R/S or

W/E lock request, a node generates only one message (the minimum number). When a node releases a lock, only one message is generated (the minimum number). No additional messages are generated for the lock actually to be granted to the next requester since the nodes are monitoring the request and release traffic to determine when they should be granted the lock. For the barrier operation, one message is generated to initialize the barrier, which is the minimum. One message is generated for each node arriving at the barrier, which is the minimum. No additional messages are generated when barrier completion occurs since each node calculates for itself when barrier completion occurs. For the F&OP command, each node generates a message when issuing the command, which is the minimum. No additional messages are needed for the node to calculate the result of its F&OP or keep the the target `V` updated since each node observes all the command traffic and performs those calculations locally.

Together, the third and fourth observations demonstrate that the implementation of these synchronization primitives are scalable in both space and time requirements.

3.2 Implementation-dependent comments

In [1] we assumed an optical fiber implementation of the broadcast ring; this assumption allows us to make two implementation-dependent observations. First, the previous discussion has implicitly assumed that all of a given synchronization command's traffic is carried on only one broadcast ring. The high bandwidth of optical fibers (on the order of ten Terahertz) makes it possible to have, as the optical technology matures, large numbers (on the order of a hundred) of high speed (Gbit or more) channels. Thus, if the lock command (or barrier or F&OP command) traffic is heavy enough, one might allocate several high speed channels to alleviate the communication bottleneck. The algorithms previously described do not need modification to work properly if the command traffic is split over several channels as long as one restriction is observed. That restriction is that all command traffic pertaining to one instance of a given synchronization command must be carried on one channel. An additional implication of splitting a given type of command traffic over several channels is that the node will need to replicate the hardware that implements the monitoring algorithms.

As a second observation (that assumes using optical fibers), we show some minimum

and maximum execution times of these synchronization primitives for various system configurations (the details of the derivation of these times are given in [1]). Tables 1, 2, and 3 list these execution times for idle and busy locks, last arrival and simultaneous arrival barriers, and single and multiple F&OP commands respectively. These execution times are listed for small and large system sizes (50 and 100 nodes respectively) and for small and large physical scales (end-to-end propagation times of 50 nsec, or 10 m, and 500 nsec, or 100 m, respectively). An idle lock is one not being held when the node requests it; a busy lock is one for which the node waits before being granted it. A last arrival barrier is one for which all nodes but one have previously arrived; a simultaneous arrival barrier is one for which some arbitrary number of nodes arrive together at the barrier. A single F&OP command situation is one in which only one node issues the F&OP for a particular target `V`; in the multiple F&OP command situation, some number of nodes P issue the F&OP command to the same target `V`. These execution times indicate that even for large system sizes and large physical scales, the raw execution times of these three synchronization primitives is on the order of from a few microseconds to a few tens of microseconds.

4 Conclusions

We have presented efficient implementation techniques for some well-known synchronization operations that rely on local decisions for determining the outcome of such operations. We have shown that our implementation incurs exactly the minimum number of messages to perform these operations. Further, the space requirement at each node for implementing these algorithms is a constant independent of the number of participating processors. Thus our implementation is scalable in terms of both time and space requirements. These algorithms depend upon the interconnection network being a broadcast ring having three particular properties. These algorithms demonstrate that it is useful to consider how an interconnection network's structure (and resulting properties) can aid the implementation of useful parallel programming constructs. By assuming a particular implementation technology for the broadcast ring (optical fibers), we have also presented some results showing the raw execution performance of these commands to be reasonable over even large system sizes and physical scales. The next research step is to consider the various practical issues

and concomitant ramifications of constructing systems using these techniques. There are several different questions to be addressed in this next step. What are the engineering issues in building the broadcast ring in either electronic or optical hardware? How many nodes can be supported in either technology? What are the design issues involved in storing the replicated data structures distinct from the architecture's shared memory? Should local hardware tables be used for this storage? Are there existing parallel systems (e.g., the KSR machine [3]) having a ring-style interconnection network which could utilize these algorithms? Are there other interconnection network structures which can be utilized in a similar manner? Finally, how do "real" parallel programs perform when using these synchronization primitives' implementations versus using other (previously suggested) software and hardware implementations?

References

[1] M. H. Davis, Jr. *Optical Waveguides in General Purpose Parallel Computers.* PhD thesis, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280, Dec. 1992. Available as Technical Report GIT-CC-93/06.

[2] A. Gottlieb and C. Kruskal. Coordinating parallel processors: A partial unification. *Computer Architecture News*, pages 16–24, Oct. 1981.

[3] Kendall Square Research Corp., Waltham, Massachusetts. *Kendall Square Research Technical Summary*, 1992.

[4] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

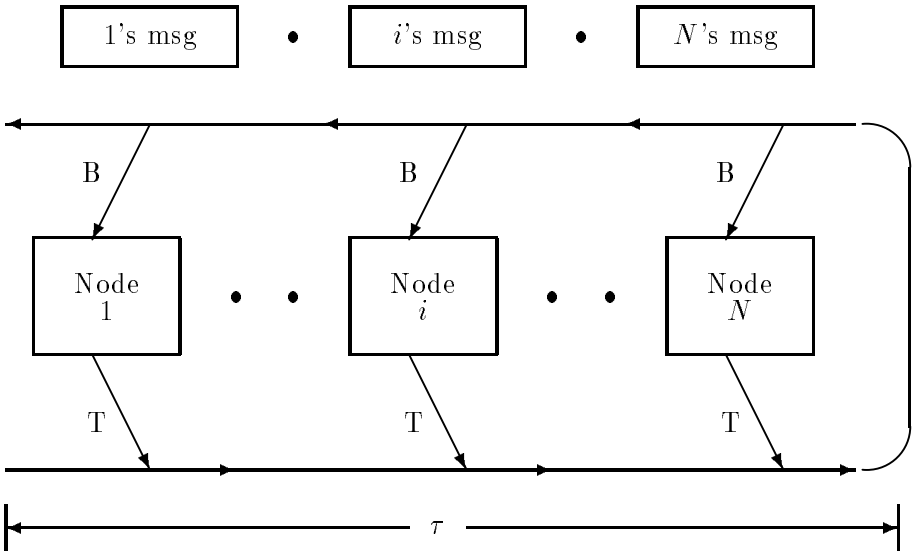

Figure 1: The broadcast ring's properties.

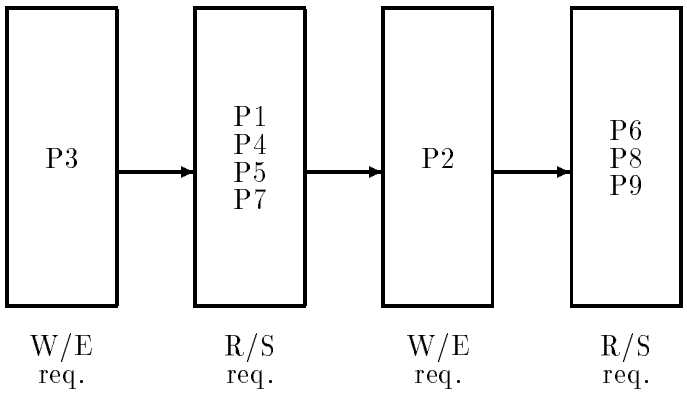

Figure 2: Lock request peer groups.

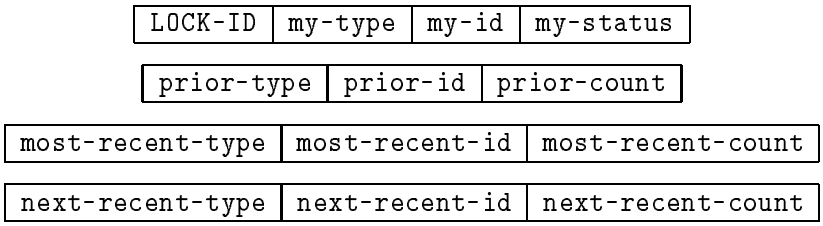

Figure 3: The data structure associated with each lock.

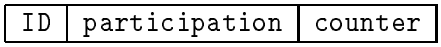

Figure 4: The data structure which represents the state of each barrier.

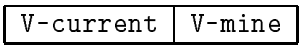

Figure 5: The data structure associated with each target `V`.

N	τ (nsec)	Execution Time (nsec)			
		Idle		Busy	
		min	max	min	max
50	50	70	2555	170	3910
50	500	70	2555	670	4360
500	50	70	25055	170	37660
500	500	70	25055	1070	38110

Table 1: Minimum and maximum execution times for idle and busy locks. N is the system size, and τ is the physical scale.

N	τ (nsec)	Execution Time (nsec)			
		Last		Simul	
		min	max	min	max
50	50	70	2605	670	2655
50	500	70	3055	670	3555
500	50	70	25105	6295	25155
500	500	70	25555	6295	26055

Table 2: Minimum and maximum execution times for last arrival and simultaneous arrival barriers. N is the system size, and τ is the physical scale.

N	τ (nsec)	Execution Time (nsec)			
		Single		Multiple	
		min	max	min	max
50	50	70	2555	170	2555
50	500	70	2555	670	2555
500	50	70	25055	170	25055
500	500	70	25055	1070	25055

Table 3: Minimum and maximum execution times for single command and multiple command F&OP. N is the system size, and τ is the physical scale.

A Lock Monitoring Algorithms

```
if (COMMAND == REQUEST):
  if (SENDER != SELF):  //  Just update  most-recent and
                            next-recent peer groups.
    switch REQUEST_TYPE:
     case  W/E: //  Automatically starts new Peer Group.
        next-recent-type   := most-recent-type;
        next-recent-id     := most-recent-id;
        next-recent-count  := most-recent-count;

        most-recent-type   :=  W/E;
        most-recent-id     := REQUEST-id;
        most-recent-count  := 1;

        break;

     case  R/S:
        if (most-recent-type == ( NONE .OR.  W/E)):
           //  Starts a new Peer Group.
           next-recent-type   := most-recent-type;
           next-recent-id     := next-recent-id;
           next-recent-count  := next-recent-count;

           most-recent-type   :=  R/S;
           most-recent-id     := REQUEST-id;
           most-recent-count  := 1;

        else:  //  Since  most-recent Peer Group is R/S, this request
                   joins it.
           most-recent-count++;
           endif.

        break;
     endswitch.
    endif.
  endif.
```

Figure 6: The algorithm which nodes follow for processing Lock Requests from other nodes.

```
if (COMMAND == REQUEST):
  if (SENDER == SELF):  // Must define my and prior
                           peer groups, as well as update most-recent
                           and next-recent peer groups.
    switch REQUEST_TYPE:
     case W/E: //  Automatically starts new Peer Group.
        //  First create prior group from most-recent group:
        prior-type   := most-recent-type;
        prior-id     := most-recent-id;
        prior-count  := most-recent-count;

        //  Update next-recent and most-recent groups:
        next-recent-type   := most-recent-type;
        next-recent-id     := most-recent-id;
        next-recent-count  := most-recent-count;

        most-recent-type   :=  W/E;
        most-recent-id     := REQUEST-id;
        most-recent-count  := 1;

        //  Create my group:
        my-type  := most-recent-type;
        my-id    := most-recent-id;

        if (prior-type == NONE): my-status :=  GRANTED;
        else:                    my-status :=  PENDING; endif.

        break;
```

First part (of three) of Fig. 7.

Figure 7: The algorithm which nodes follow for processing Lock Requests from themselves.

```
case R/S:
   if (most-recent-type == ( NONE .OR. W/E):
     // Form new Peer Group:
     // First create prior group from most-recent:
     prior-type   := most-recent-type;
     prior-id     := most-recent-id;
     prior-count  := most-recent-count;

     // Update next-recent and most-recent groups:
     next-recent-type   := most-recent-type;
     next-recent-id     := most-recent-id;
     next-recent-count  := most-recent-count;

     most-recent-type   := R/S;
     most-recent-id     := REQUEST-id;
     most-recent-count  := 1;

     // Create my group:
     my-type  := most-recent-type;
     my-id    := most-recent-id;

     if (prior-type == NONE): my-status := GRANTED;
     else:                    my-status := PENDING; endif.
```

Second part (of three) of Fig. 7.

```
        else: //  Since  most recent Peer Group is R/S, this request
              //  joins it.
              //  First create  my group:
          prior-type   := next-recent-type;
          prior-id     := next-recent-id;
          prior-count  := next-recent-count;

          //  Update  most recent group:
          most-recent-count++;

          //  Create  my group:
          my-type := most-recent-type;
          my-id   := most-recent-id;

          if (prior-type ==  NONE): my-status :=  GRANTED;
          else:                     my-status :=  PENDING; endif.

        endif.
       break;
     endswitch.
    endif.
  endif.
```

Third part (of three) of Fig. 7.

```
if (COMMAND ==  W/E-UNLOCK):
 if (REQUEST-id == next-recent-id): next-recent-type :=  NONE; endif.

 if (REQUEST-id == most-recent-id): most-recent-type :=  NONE; endif.

 if (REQUEST-id == prior-id):
   prior-type :=  NONE;
   if (my-type == ( W/E .OR.  R/S):
     my-status :=  GRANTED;
     endif.
   endif.

 if (REQUEST-id == my-id): //  Change  my-type to indicate that node
                           //  no longer holds this lock—there is no
                           //   my-count field to update.
   my-type : =  NONE;
   endif.

 endif.
```

Figure 8: The algorithm which nodes follow for processing W/E UNLOCK commands.

```
if (COMMAND ==  R/S-UNLOCK):
  if (REQUEST-id == next-recent-id):
    next-recent-count--;
    if (next-recent-count == 0): next-recent-type :=  NONE; endif.
    endif.

  if (REQUEST-id == most-recent-id):
    most-recent-count--;
    if (most-recent-count == 0): most-recent-type :=  NONE; endif.
    endif.

  if (REQUEST-id == prior-id):
    prior-count--;
    if (prior-count == 0):
      prior-type :=  NONE;
      if (my-type == ( W/E .OR.  R/S): my_status :=  GRANTED; endif.
      endif.
    endif.

  if (REQUEST-id == my-id): //  Change  my-type to indicate that node
                            //  no longer holds this lock—there is no
                            //   my-count field to update.
    my-type : =  NONE;
    endif.

  endif.
```

Figure 9: The algorithm which nodes follow for processing R/S UNLOCK commands.

B Barrier monitoring algorithms

```
switch command:
  case INITIALIZE:
      participation := N;
      counter       := 0;
      break;

  case REACH:
      counter++;
      break;

      endcase;
   endswitch.
```

Figure 10: The monitoring algorithm that each node follows in processing barrier commands.

C F&OP monitoring algorithms

```
if (SENDER == SELF):
  V-mine := V-current;
   notify CPU that the F&OP has completed by returning V-mine;
  endif;

V-current := (V-current OP e).
```

Figure 11: The monitoring algorithm each node follows in the hybrid F&OP scheme.