

Mutual Exclusion on Optical Buses

Krishna M. Kavi

Electrical and Computer Eng Dept
University of Alabama in Huntsville
Huntsville, AL 35899

Dinesh. P. Mehta

Dept of Math and Computer Science
Colorado School of Mines
Golden, CO 80401

Abstract

This paper presents two algorithms for mutual exclusion on optical bus architectures including the folded one-dimensional bus, the one-dimensional array with pipelined buses (1D APPB), and the two-dimensional array with pipelined buses (2D APPB). The first algorithm guarantees mutual exclusion, while the second guarantees both mutual exclusion and fairness. Both algorithms exploit the predictability of propagation delays in optical buses.

1 Introduction

Fiber optics based communication technologies have been utilized in Wide Area Networks because they offer higher bandwidths and lower error probability than other communication technologies. More recently, optical properties such as unidirectional propagation and predictable propagation delays have been touted as beneficial in building massively parallel processing systems using optical buses for interconnection among processing nodes. Time Division Multiplexing (TDM) enables the implementation of pipelined optical buses [1]. Alternatively, Wavelength Division Multiplexing (WDM) can be used to create multiple channels on which multiple messages can simultaneously be transmitted. The multiple channels can be either statically or dynamically allocated to processors [2]. Utilizing the ability to transmit multiple messages simultaneously on optical buses (in pipelined fashion or using multiple channels), researchers have described efficient algorithms for parallel computing based on the message-passing paradigm (See [3] for an excellent survey). To the best of our knowledge, the shared-memory paradigm for parallel computing has not been investigated for its suitability on optically interconnected parallel processing systems. In this paper we describe how mutual exclusion can be implemented on pipelined optical buses. The problem of mutual exclusion is critical to the shared-memory paradigm of parallel computation since it arises whenever concurrent access to shared resources by several

sites is involved [4]. For correctness it is necessary to guarantee that the shared resource is accessed by only one site at a time (i.e., mutual exclusion). In addition to assuring mutual exclusion, techniques for mutual exclusion in distributed systems must exhibit the following characteristics: (1) *Freedom from deadlocks*: Two or more sites should not endlessly wait for events that will never occur. An event can be the arrival of a message. (2) *Freedom from starvation*: A site should not be forced to wait indefinitely to acquire a shared resource, while other sites are repeatedly acquiring the resource. In other words, any site should be allowed to request and acquire a shared resource in a finite amount of time. (3) *Fairness*: Fairness dictates that requests must be executed in the order they are made, or in the order they arrive in the system.

In this paper we will assume Time Division Multiplexing based optical wave guides that facilitate unidirectional pipelined buses. Our work is based on two related models for systems that utilize pipelined buses: Linear array with a reconfigurable pipelined bus system (LARPBS) [1, 5] and array with reconfigurable buses (AROB) [6]. Many parallel algorithms were proposed using these models. In LARPBS counting is not allowed during a bus cycle. In fact, the LARPBS does not allow any processing during a bus cycle, except for setting switches at the beginning of a bus cycle. AROB allows counting during a cycle. More detailed description of the functional behavior of such pipelined buses, and how “coincident pulse” methods can be used to achieve point-to-point, multicast and broadcast data communications can be found in [5].

In this paper, we will make the following extensions to the pipelined bus models: (1) *Bus cycle*: Most researchers define a bus cycle as the time needed for a message to travel the entire length of the bus. For our purposes, we define a cycle as the amount time needed for a message to travel one segment of the pipelined bus (that is, the time to travel between two adjacent sites). (2) *Computation during a bus cycle*: Atomicity which is fundamental to the implementation of mutual

exclusion requires that computation be performed as events occur. This implies that delay loops may be required in pipelined optical bus segments to accommodate computations required by the techniques described in this paper. In other words, we will assume that during a cycle, an optical message travels one segment of the bus, and a processing node performs some computations when a message is received. Alternately, we can require two phases: Communication and Computation. During communication phase, processors transmit (and receive) messages on the optical pipelined bus. At most one message can be placed on the bus by a processor. All messages move along the bus synchronously. The communication phase is complete only when (all) messages travel the entire length of the bus assuring receipt by all processors. During the computation phase, processors examine the messages received during the communication phase, and perform computations as needed to achieve mutual exclusion. The system operates with alternating communication and computation phases. To simplify this paper, we will assume that bus cycle includes computation. However, our techniques will also work in the second case with alternating communication and computation cycles. (3) *Bus contention*: Since we include computation in a bus cycle, we permit asynchronous and simultaneous requests from sites. In other words, processors are NOT required to transmit messages synchronously, and they can place a request at any time. This can lead to bus contention as the messages move down the pipelined buses, if new message requests are inserted by other sites. In this paper we will assume that the buses are designed to eliminate such collisions on the bus and a processor excludes itself from placing a new message if that message collides with a message that is already on the pipelined bus. This assumption is not needed in the alternating communication-computation phases model described above.

2 Optical Bus Architectures

2.1 The Folded One-Dimensional Bus

The folded one-dimensional bus model[7, 8] is shown in Figure 1. It consists of a sequence of n equidistant processors connected by a folded bus. A processor is connected to both the upper and lower segments of the bus. A processor transmits messages on the upper segment of a bus and receives messages from the lower segment. In Figure 1, a message originates on the upper segment and travels in the direction of the arrow. Assume that it takes 1 unit of time for a message to travel from one processor to its neigh-

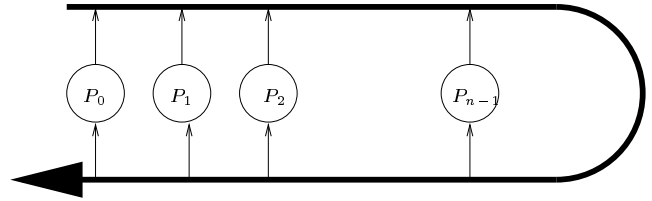


Figure 1: Folded Unidirectional Bus Architecture

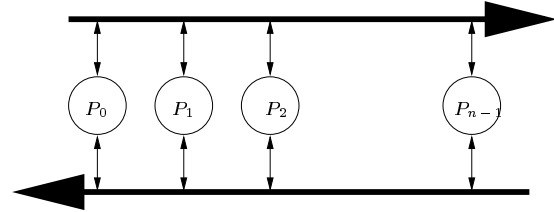


Figure 2: Unfolded Unidirectional Bus Architecture

bor. This architecture permits a pipeline of messages to simultaneously coexist on an optical bus. Therefore, processors P_0 and P_1 can place messages on the bus at the same instant of time. An arbitrary processor P_i receives P_0 's message exactly 1 time unit after it receives P_1 's message. In general, a message from processor P_i to P_j takes $d_{ij} = 2n - 1 - i - j$ time units. As stated earlier, it is assumed that the optical transmission hardware on the upper segment is capable of conditionally transmitting a message so that this will not cause a collision. This will prevent P_i from transmitting at time t if processor P_{i-1} transmitted at time $t - 1$.

2.2 The One-Dimensional Array with Pipelined Buses (1D APPB)

The 1D APPB is shown in Figure 2. This is similar to the folded bus above. Here processors can transmit and receive on either segment. For example, a message from P_0 to P_2 would be sent on the upper segment, while a message from P_2 to P_0 would be sent on the lower segment. Once again, we assume that processors have the capability to conditionally transmit a message so that there are no bus collisions. In this model, a message from P_i to P_j takes $d_{ij} = |i - j|$ time units.

3 Mutual Exclusion Algorithms

Consider a multiprocessing environment where processors are in contention for a particular resource (e.g., a particular word in memory). When this happens, we would like to ensure that at most one processor has access to the resource at any given instant. We achieve

this below by assigning a unique integer to each contending processor that indicates the order in which it must access that resource. This integer will be referred to as the processor’s “turn”. This contention phase is followed by an access phase where each processor actually accesses the resource in the prescribed order. When a processor has completed its access, it broadcasts a message to this effect to the other processors and the next processor proceeds to access the resource. The key to implementing this philosophy is to correctly design the algorithm that is to be employed during the contention phase so that different contending processors are guaranteed to be assigned different turns.

A processor P_i first communicates its request for a given resource to all of the other processors. When processor P_j receives P_i ’s request, it immediately places a hold on any future requests that it might have for that resource until P_i ’s turn is confirmed. The difficulty arises when P_j places its request for the same resource before it receives P_i ’s request. The purpose of a mutual exclusion algorithm is to ensure that this situation is resolved in a consistent fashion. We next present two mutual exclusion algorithms for optical bus architectures. These algorithms are presented in an interconnection architecture-independent fashion.

3.1 A Window-Based Algorithm

Define the *diameter* D of a network to be the longest delay (or time taken for a message to travel) between any pair of processors in the network; i.e., $D = \max_{ij, i \neq j} d_{ij}$, where d_{ij} is the delay (as defined in Sections 2.1 and 2.2) between P_i and P_j . Note that the diameter depends on the specific optical bus architecture that is being considered. Next, we define the *window of vulnerability* for each processor to be twice the diameter of the network; i.e., $2D$.

We begin by outlining some assumptions and principles on which our algorithms are based.

1. *Rule 1*: A resource request is a message that contains the *id* of the processor making the request and the resource being requested.
2. *Rule 2*: When a processor *makes* a request for a resource, it may not make additional requests until the window of vulnerability for its request has elapsed.
3. *Rule 3*: A processor P_j that *receives* a request for a resource from P_i delays making a future request for that resource until P_j ’s window of vulnerability has expired.

4. *Rule 4*: When requests from several processors overlap in time, processor priorities are used to resolve the requests. Without loss of generality, we assume that a lower numbered processor has higher priority (i.e., P_0 has the highest priority).

Our mutual exclusion algorithm requires certain actions to be taken by a processor when it receives a resource-request message and when it wishes to send a resource-request. Algorithm *Receive_Request* (Figure 3) below describes the actions to be taken by a processor i when it receives a request for a resource at time t . For convenience, we consider the current time step (t) and the processor’s ID (i) to be arguments to the function. The Resource_ID parameter (r) and the Time parameter (t_i) are assumed to be NULL if P_i has no outstanding requests at time t . If P_i does have outstanding requests at time t , then these quantities denote the resource that was requested and the time at which the request was sent.

```

Receive_Request(TIME  $t$ , Proc_ID  $i$ , Resource_ID  $r$ , Time  $t_i$ )
begin
  if (a resource request msg is received by  $P_i$  at time  $t$ )
    // assume that requests originating at  $P_i$  are ignored.
    begin
      RECV(message);
      Processor_ID  $j$  = message.getProcID();
      Resource_ID  $s$  = message.getResourceID();

      if ( $r = s$  and  $j < i$ )  $my\_turn[r]++$ ; // Rule 4
       $next\_turn[r]++$ ;

      //Ensure that  $P_i$  sends any future requests for  $s$ 
      //after  $j$ ’s window of vulnerability has expired;
      // i.e., guarantees Rule 3
      if ( $r \neq s$ )
        begin
           $earliest[s] = \max(t - d_{ji} + 2D + 1, earliest[s])$ ;
           $next\_turn[s]++$ ;
        end
      end
    end
end

```

Figure 3: Algorithm *Receive_Request*.

Note that $t \leq t_i + 2D$ must be true if P_i has an outstanding request (*Rule 2*). The resource table (array my_turn) is contained in each processor. It is assumed that these arrays initially contain the same values in each processor. The variable $my_turn[r]$ for a resource r in processor P_i denotes the order in which processor P_i will get access to resource r . Thus, the objective

is that when several processors request the same resource r , our mutual exclusion algorithm must ensure that each of these processors has a different local value of $my_turn[r]$ that indicates the order in which that processor will access the resource.

When a processor wishes to make a request for a resource r , it first checks that the current time is greater than that in $earliest[r]$ and if this is true, sets $my_turn[r]$ to $next_turn[r]$ and increments $next_turn[r]$. Figure 4 contains the details.

```

Send_Request(TIME  $t$ , Processor_ID  $i$ , Resource_ID  $r$ )
begin
  if ( $earliest[r] > t$ )
    Wait until  $earliest[r]$  to send request;
  else
    begin
      SEND(request);
       $my\_turn[r] = next\_turn[r]$ ;
       $next\_turn[r]++$ ;
       $earliest[r] = t + 2D + 1$ ; // Rule 2
    end
end

```

Figure 4: Algorithm *Send_Request*.

Example: Consider a folded unidirectional bus architecture (Fig 1) with $n = 10$ processors. Suppose that processors P_0 and P_2 send resource requests at time 0, P_7 at time 8, P_1 at time 11, P_9 at time 16, and P_8 at time 22. Assume that all requests are for the same resource. We describe the step-by-step operation of our algorithms using Table 1.

At time 0, P_0 and P_2 send request messages. As per function *Send_Request*, both processors set their my_turn variables to 0, increment their $next_turn$ variables, and initialize $earliest$ to $2D + 1 = 37$. In the next 7 time steps, their messages travel towards the right on the upper segment of the folded bus. At time step 8, P_2 's message becomes available at P_9 's receiver. This causes P_9 to update its $next_turn$ and $earliest$ variables as outlined in function *Receive_Request*. Simultaneously, in step 8, P_7 sends a request. In steps 9 through 17, P_2 's message travels towards the left in the lower segment causing variables in all the processors to be updated. Each processor increments $next_turn$ and, if necessary, updates $earliest$. Also, observe, that P_7 increments its my_turn variable in step 10 since it has a lower priority than P_2 . However, P_0 does not change its turn in step 17 since it has higher priority than P_2 . In the mean time, P_1 sends a resource request in step

11 and its $next_turn$ variable is subsequently updated when P_2 's message passes through in step 16. The messages from P_0, P_7 , and P_1 make their appearances at P_9 at steps 10, 11, and 20, respectively. In subsequent steps, these messages travel towards the left in the lower segment updating variables as specified in *Receive_Request*. Notice that when a message passes the processor from which it originated, no change is made to the processor's variables (e.g., P_2 in step 15, P_0 in step 19, P_7 in step 13, and P_1 in step 28). Finally, we observe that the messages from P_9 and P_8 , which were to be sent at steps 16 and 22, respectively, never get sent in our snapshot! This is because the value of the $earliest$ variable in those processors at the specified times are greater than the time step (e.g., the $earliest$ variable at step 16 in P_9 is 44 and $44 > 16$). Observe, that on completion, all processors have a local $next_turn$ value of 4 and an $earliest$ value of 48. The four processors that got their requests out before receiving any requests (i.e., P_0, P_1, P_2 , and P_7) have been given unique turns according to their priorities.

Lemma 1 *If the windows of vulnerability for two processors overlap, then both processors are scheduled consistently with respect to each other.*

Proof Let P_i and P_j be two processors whose windows overlap. Without loss of generality, assume that P_i makes its request first at time t_i . Since P_j made an overlapping request, the request must have been made at time t_j such that $t_i \leq t_j \leq t_i + d_{ij}$ (Rule 3). P_j 's request reaches processor P_i at time $t_j + d_{ji} \leq t_i + d_{ij} + d_{ji} \leq t_i + 2D$. Therefore, P_i receives P_j 's request in its (P_i 's) window of vulnerability and vice versa. Since processors are prioritized consistently throughout the network, the turns allocated to each processor are consistent with respect to each other; e.g., if $j < i$, then P_i 's turn will be incremented whereas P_j 's turn will remain the same. \square

Theorem 1 *The algorithm presented above guarantees mutual exclusion.*

Proof We show that different processors requesting the same resource will be assigned a "turn" for that resource such that each processor is assigned a different turn. First, we show that any set S_j of requests for resource j over some period can be partitioned into subsets $S_j^1, S_j^2, \dots, S_j^k$, such that the windows of vulnerability corresponding to set S_j^k all overlap with each other and do not overlap with windows of vulnerability from any $S_j^l, l \neq k$. If this is not true, there must exist a triple of requests from P_i, P_j , and P_k such that

Time	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
< 0	(0,-,-)	(0,-,-)	(0,-,-)	(0,-,-)	(0,-,-)	(0,-,-)	(0,-,-)	(0,-,-)	(0,-,-)	(0,-,-)
0 - 7	(1,0,37)	—	(1,0,37)	—	—	—	—	—	—	—
8	—	—	—	—	—	—	—	(1,0,44)	—	(1,-,37)
9	—	—	—	—	—	—	—	—	(1,-,37)	—
10	—	—	—	—	—	—	—	(2,1,44)	—	(2,-,37)
11	—	(1,0,48)	—	—	—	—	(1,-,37)	—	(2,-,37)	(3,-,44)
12	—	—	—	—	—	(1,-,37)	—	(3,2,44)	(3,-,44)	—
13	—	—	—	—	(1,-,37)	—	(2,-,37)	—	—	—
14	—	—	—	(1,-,37)	—	(2,-,37)	(3,-,44)	—	—	—
15	—	—	—	—	(2,-,37)	(3,-,44)	—	—	—	—
16	—	(2,0,48)	—	(2,-,37)	(3,-,44)	—	—	—	—	—
17	(2,0,37)	—	(2,1,37)	(3,-,44)	—	—	—	—	—	—
18	—	(3,1,48)	(3,1,44)	—	—	—	—	—	—	—
19	—	(4,1,48)	—	—	—	—	—	—	—	—
20	(3,0,37)	—	—	—	—	—	—	—	—	(4,-,48)
21	—	—	—	—	—	—	—	—	(4,-,48)	—
22	—	—	—	—	—	—	—	(4,3,48)	—	—
23	—	—	—	—	—	—	(4,-,48)	—	—	—
24	—	—	—	—	—	(4,-,48)	—	—	—	—
25	—	—	—	—	(4,-,48)	—	—	—	—	—
26	—	—	—	(4,-,48)	—	—	—	—	—	—
27	—	—	(4,2,48)	—	—	—	—	—	—	—
28	—	—	—	—	—	—	—	—	—	—
29	(4,0,48)	(4,1,48)	(4,2,48)	(4,-,48)	(4,-,48)	(4,-,48)	(4,-,48)	(4,3,48)	(4,-,48)	(4,-,48)

Table 1: An illustration of the values of the variables in each processor as a function of time. Each table entry is a triple consisting of the *next_turn*, *my_turn*, and *earliest* variables (in that order) contained in the given processor at a given instant of time. A “—” indicates that there was no change in the variables belonging to a processor relative to the previous time step.

(i) P_i and P_j 's windows overlap. (ii) P_j and P_k 's windows overlap. (iii) P_i and P_k 's windows do not overlap. Without loss of generality, assume that $t_i < t_j < t_k$. Note that $t_j \leq t_i + d_{ij}$ and $t_k \leq t_j + d_{jk}$ implying that $t_k \leq t_i + d_{ij} + d_{jk} \leq t_i + 2D$. Thus, P_i and P_k 's windows do overlap, and the hypothesis is proved by contradiction. Since all the requests in subset S_j^k have mutually overlapping windows, they are all scheduled consistently with respect to each other (Lemma 1). \square

3.2 A Timestamp-Based Algorithm

Since priorities are hard-wired into the algorithm of the previous section, the mutual exclusion protocol described in the previous section is consistently biased against low priority (i.e., higher numbered) processors. Thus, the algorithm is not fair since, even if the request from a lower priority site originated earlier than that of a higher priority site, the higher priority site may be granted the mutual exclusion request before the lower priority site. For example, every time P_i and P_j , $i < j$, place requests so that their windows overlap, P_i gets access to the resource before P_j even if $t_j < t_i$. In this section, we present an algorithm that operates on a First Come First Served (FCFS) basis; i.e., if P_j places a request before P_i , it gets earlier access to the resource even if their windows overlap. We revert to the priority scheme of the previous section in the rel-

atively unlikely event that several processors place a request at the same time. Algorithm *Receive_Request* (Figure 5) utilizes the predictability of delays in an optical bus to determine the time of request of a message based on the time that the message is received at a processor. For example, if P_i receives a request message from P_j at time t , P_j 's message must have originated at time $t - d_{ji}$. If this quantity is less than t_i , the time at which P_i sent its message, P_j gets access to the resource before P_i .

Theorem 2 *The algorithms presented above guarantee mutual exclusion and fairness.*

Proof Mutual exclusion is guaranteed by reasoning identical to that in Theorem 1. The timestamp technique described above clearly causes requests to be handled in an FCFS fashion. \square

We note that the algorithms can be easily extrapolated to other optical networks by choosing appropriate values for d_{ij} .

3.3 Performance Analysis

In this section, we analyze the performance of the *Timestamp* algorithm of the previous section assuming the folded bus architecture. We begin by defining some performance measures for mutual exclusion algorithms. Our definitions are adapted from those in [4].

```

Receive_Request(TIME  $t$ , Proc_ID  $i$ , Resource_ID  $r$ , Time  $t_i$ )
begin
  if (there is a resource req msg at  $P_i$ 's receiver at time  $t$ )
  // assume that requests originating at  $P_i$  are ignored.
  begin
    RECV(message);
    Processor_ID  $j$  = message.getProcID();
    Resource_ID  $s$  = message.getResourceID();

    if ( $r = s$ ) then
    begin
      if ( $t - d_{ji} < t_i$ ) then  $my\_turn[r]++$ ;
      elseif ( $t - d_{ji} = t_i$  and  $j < i$ )
      then  $my\_turn[r]++$ ;
       $next\_turn[r]++$ ;
    end
    else //  $r \neq s$ 
    // Ensure that  $P_i$  sends any future requests for  $s$ 
    // After  $j$ 's window of vulnerability has expired.
    begin
       $earliest[s] = \max(t - d_{ji} + 2D + 1, earliest[s])$ ;
       $next\_turn[s]++$ ;
    end
  end
end
end

```

Figure 5: Algorithm *Receive_Request*.

In order to determine the synchronization delay SD , we assume that when a processor releases a lock, it broadcasts a release message to all the other processors. On a folded bus, this takes a minimum of n units (for P_{n-1}) and a maximum of $2n - 2$ units (for P_0 or P_1). To determine the throughput TH , we assume that a processor spends E time units in a critical section. Then, $TH = 1/(SD + E)$. Appropriate values for SD can be substituted to obtain minimum and maximum throughput. Next, we present best and worst case analyses for response time RT for some arbitrary processor P_i . The best case is when the lock is not held by any other processor and no other processor requests the lock during P_i 's window of vulnerability. In this case, $RT = 2D + E$. Alternatively, if we make the common assumption that in the best case, P_i has to wait for a single processor P_j to release the lock, RT also includes the remaining execution time of P_j ($E/2$ on the average) and the synchronization delay after P_j releases the lock (approximately $3n/2$ on the average). Then $RT = 2D + 3/2E + 3n/2$. In the worst case, all processors request the resource at the same instant of time and P_i 's request receives the low-

est priority (i.e., $i = n - 1$). So, after waiting for the window of vulnerability to expire, P_{n-1} must wait for the remaining $n - 1$ processors to execute and to incur their synchronization delays before performing its own execution. Here,

$$\begin{aligned}
 RT &= 2D + nE + 2n - 2 + \sum_{i=1}^{n-2} (2n - i - 1) \\
 &= 2D + nE + 3/2n^2 - 3/2n - 1
 \end{aligned}$$

Note that P_0 has $SD = 2n - 2$, while an arbitrary P_i has $SD = 2n - i - 1$.

References

- [1] K. Li, Y. Pan, S.Q. Zheng, "Pipelined TDM optical bus with conditional delays," *Optical Engineering*, vol. 36, pp. 2417-2424, Sept. 1997.
- [2] D.C. Hoffmeister et al, "Lightning network and system architecture," in *Parallel Computing using Optical Interconnections* (K. Li, Y. Pan, S.Q. Zheng, ed.), Kluwer Academic Press, 1998.
- [3] K. Li, Y. Pan, S.Q. Zheng, ed., *Parallel Computing using Optical Interconnections*. Kluwer Academic Press, 1998.
- [4] M. Singhal, N. Shivaratri, *Advanced Concepts in Operating Systems*. New York: McGraw-Hill, 1994.
- [5] Y. Pan, "Basic Data Movement Operations on the LARPBS Model," in *Parallel Computing using Optical Interconnections* (K. Li, Y. Pan, S.Q. Zheng, ed.), Kluwer Academic Press, 1998.
- [6] S. Pavel, S.G. Akl, "Computing the Hough Transform on Arrays with Reconfigurable Optical Buses," in *Parallel Computing using Optical Interconnections* (K. Li, Y. Pan, S.Q. Zheng, ed.), Kluwer Academic Press, 1998.
- [7] Z. Guo et al, "Pipelined communication in optically connected arrays," *Journal of Parallel and Distributed Computing*, vol. 12, no. 3, pp. 269-282, 1991.
- [8] C. Qiao, R. Melhem, "Time-Division Optical Communications in Multiprocessor Arrays," *IEEE Transactions on Computers*, vol. 42, pp. 577-590, May 1993.