Conservative Parallel Simulation of Finite Buffered Multistage Interconnection Networks

Seng Chuan TAY and Yong Meng TEO
Department of Information Systems and Computer Science
National University of Singapore
Kent Ridge, SINGAPORE 0511
email: {taysengc,teoym}@iscs.nus.sg

Abstract

Multistage interconnection networks are used in a number of application areas such as parallel computers and high-speed communication systems. As the performance of these systems lies on an efficient design of the interconnection network, a thorough analysis of the network's performance is important. Mathematical analysis so far provides inadequate results and simulation analysis using a uniprocessor usually requires long hours to evaluate large networks. In this paper, parallel simulation technique is used to speedup the execution. The conventional null-message approach to resolving deadlock problem in conservative simulation is based on a lookahead mechanism. For some application domains, unfortunately, the lookahead information is not available. Consequently, parallel simulation using null messages can result in livelock. We propose a deadlock/livelock free scheme using null messages, but without the guaranteed lookahead, to coordinate the simulation. In addition, we investigate different partitioning and transformation techniques for mapping a simulation program onto multicomputers. A flushing mechanism to address the combinatoric explosion of using null-message in conservative simulation is also discussed. Our analysis shows that the proposed flushing mechanism effectively reduces the number of null messages from *exponential* to *linear*.

Keywords: parallel simulation, conservative approach, multistage interconnection network, deadlock, livelock

1 Introduction

Uniprocessor systems are approaching their limit on computation speed. The computation power of such systems cannot be improved indefinitely due to the fact that the speed of signal transmission on a chip is bounded by the speed of light [1]. During the last two decades there are increasing interests in multiprocessor systems consisting of hundreds, or even thousands, of processors. The communication among the processors is facilitated by message passing or by shared memory. In either types of the architecture, a communication network is required to connect the processors or to connect processors to memory modules. Multistage interconnection network (MIN) offers a tradeoff between cost and performance. More recently, there are growing interests in using MIN as interconnection structure for the switch nodes of high-speed communication networks. MIN is a potential candidate for high speed switching systems such as broadband integrated services digital network (B-ISDN), and transport systems based on the asynchronous transfer mode (ATM) [17, 18]. As the performance of these systems depends on the efficient design of the communication subsystem, a thorough analysis of MIN characteristics such as the bandwidth, latency, etc. is important. Mathematical studies for MIN's performance are widely available in the literature [10, 16, 21]. Most of the works assume that each input component generates random and independent packet transmissions distributed uniformly over all output components. So far, no closed-form result is available for non-uniform distribution [4]. Researchers also make unrealistic assumptions such as ignoring blocked transmissions [13, 16]. In the real system, this will mean that a processor may proceed with its computation before the required data is available in the processor. As mathematical analysis does not provide adequate results, the significance of simulation studies on the MIN's performance becomes apparent. Lee and Wu used simulation method to analyze the performance of circuit switching baseline networks with respect to different types of strategy to handle blocked memory requests [13]. Barnes and Lundstrom gave a simulation validation for an interconnection network used in a numerical aerodynamic simulator [2]. A simulation of buffered delta networks was carried out by Dias and Jump [5]. Since classical sequential simulation is time-consuming especially for large interconnection networks, it is a natural attempt to use multiple processors to speedup the simulation.

A promising avenue to reduce the MIN's simulation runtime is to exploit parallelism in the switching components, and map the simulation program onto multiprocessor machines. There are two main paradigms used in organizing parallel simulation: *conservative approach* and *optimistic approach* [6, 8, 15]. This paper focuses on a *conservative approach* in the parallel simulation of MINs, efficient process modeling and partitioning schemes for synchronous and asynchronous message transmissions, and an analysis of its performance. We also discuss the coordination of parallel simulation and the use of null messages to prevent deadlocks and address the combinatoric explosion of null-message overhead problem in conservative simulation. Unlike the conventional null-message approach based on a *lookahead* mechanism, we propose a deadlock/livelock free scheme, but without the guaranteed lookahead, for the MIN simulation. Section 2 gives the preliminaries, mathematical notations, states of a switching element, routing strategies and routing algorithm of MIN, and serves as a delineation for the essential capabilities of a MIN simulation. Section 3 introduces a proposed *process-oriented simulation model* for a switching element; the smallest component in a multi-stage interconnection network. We explain the methods for handling traffic contention, routing conflict resolution, parallel and preemptive message transmissions in the context of modeling and parallel simulation. We also discuss schemes for resolving forward and backward cascading deadlocks. Section 4 presents the enhanced process-oriented model for large MIN simulation, and a method to resolve the problem of circular-wait among processes. We also proposed a simple and yet effective mechanism to control the flooding of null messages during simulation. Section 5 introduces the PVM platform used to run the simulation program, investigates some partitioning and transformation techniques for mapping the simulation application onto a limited number of processors, and an analysis of the performance results. Section 6 contains our concluding remarks.

2 Multistage Interconnection Network Revisited

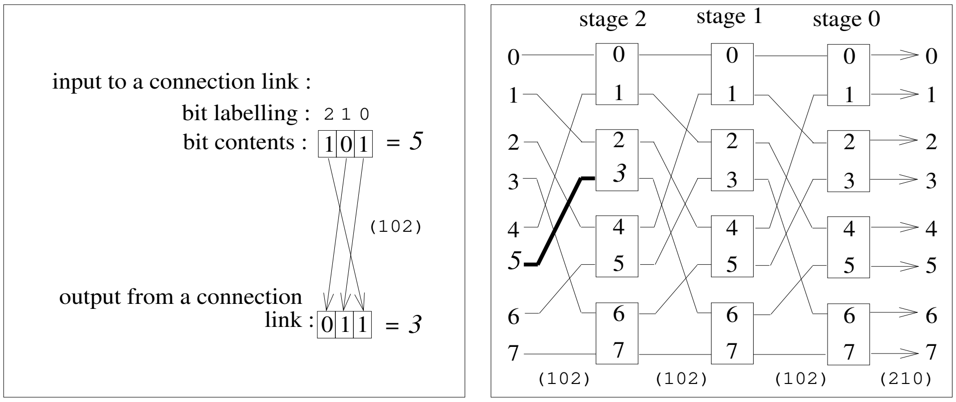
Research on interconnection networks begins as early as the existence of telephone switching systems [9]. The application of interconnection network on multiprocessor system started since the 1970s. To-date there are voluminous literature in this field. A comprehensive tutorial on MIN is available in [19, 21]. This section gives a brief description on MINs and the necessary mathematical details required in the subsequent discussions, and serves as a delineation for the essential capabilities of a MIN simulation. The Omega network is used for illustrations.

2.1 Preliminaries

A MIN (see figure 1b) consists of several stages of switching elements. For simplicity we assume that each switching element is a 2×2 crossbar, but the proposed simulation and modeling techniques can be easily generalized for switching elements of other dimensions. The input to a MIN is called *source* and output from a MIN is called *destination*. In this paper, we assume that each source is a processor and each destination can be a memory module, processor, or an I/O device. A MIN consisting of n input and m output is called a $n \times m$ MIN. For simplicity, we assume that m is equal to n. A MIN is said to have *full access capability* if every input of the network can be connected to every output of the network through the switching stages. It has been shown in [11] that for a MIN using 2×2 crossbar switching elements, the full access capability can be realized if it contains at least $\log_2 n$ stages, and with $\frac{n}{2}$ switching elements per stage. We say that a MIN is *minimum* if there exist only one path from a given source to a given destination. In other words, a minimum MIN is not fault-tolerant. Two type of switching transmissions are used in MINs, namely *blocked* and *unblocked* switchings. Traffic contention and routing conflict problems can occur in MIN's operation. To prevent any loss of information, retransmission or delayed-transmission of messages is necessary when these problems occur.

2.2 Translation Numbers

The interconnections of a MIN can be viewed as a series of permutations from the source addresses to the destination addresses, and can be represented by translation numbers as follows. First, we label



(a) Translation Number for a Connection Link

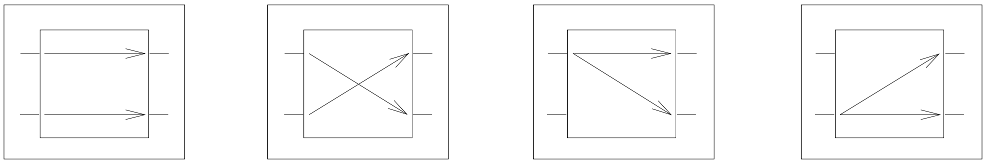
(b) 8x8 Omega Network $(102)^3(210)$

Figure 1: Representing MIN Using Translation Numbers

the input address bits starting from the least significant place. The translation number is defined as the labeling resulted after the input address bits are permuted. Take a 8×8 Omega network for example. The labeling for the input address, requiring three bits, is (210). Since the permutations for an Omega network perform a 1-bit left-rotate on the input addresses, the resulted translation number is (102) (see figure 1a). It follows that the interconnections for the 8×8 Omega network (see figure 1b) can be represented as (102)(102)(102)(210), or $(102)^3(210)$ equivalently.

2.3 States of a Switching Element

A MIN provides the connection path from a source address to a destination address by setting the appropriate states on the switching elements corresponding to the connections from the in-coming links to the out-going links as shown in figure 2. The four states are: *straight*, *swap*, *upper broadcast*, and



(a) Straight (b) Swap (c) Upper Broadcast (d) Lower Broadcast

Figure 2: Four States of a 2×2 Switching Element

lower broadcast. It is noteworthy that transmission may occur on only one connection even when two paths are established in the switching element.

2.4 Routing Strategies and Routing Algorithm

There are two main modes of communication in MINs, namely circuit switching and packet switching. In *circuit switching*, a path joining the source to its destination is established when a transmission is initiated, and is reserved until the message reaches the destination. Thus, the switching elements in the path remain in their specified state until the path is released. It is clear that this mode of communication is useful only when the amount of message transmissions is large, and the number of parallel transmissions is small. In *packet switching*, a message is broken into packets and routed through the inter-stage connections. Each packet is tagged with a destination address (and perhaps source address) and will have to establish the inter-stage path from its current stage to the next stage when it travels from source to destination. The inter-stage path and the switching element are released immediately when the packet passes the partial connection. In this paper we assume that packet switching is used. This mode of communication generates many parallel transmissions of small-size messages, rendering it suitable for parallelization.

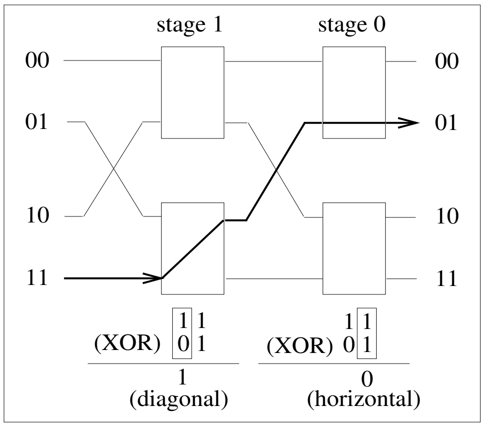


Figure 3: The Exclusive-OR Routing Tag Algorithm

There are two types of algorithm to control the message routing, i.e., *centralized* and *distributed*. Centralized control requires global information on the state of the network to establish the path connections. The second type requires a message to be tagged with a destination address, such as the *Destination Tag Algorithm* proposed by Lawrie [12]. Some distributed algorithms also require the message to be tagged with the source address, such as the *Exclusive-OR Routing Tag Technique* (XOR) [4]. In this paper, we use the XOR routing algorithm. Let the binary addresses for the source and destination be $s_{m-1}...s_1s_0$ and $d_{m-1}...d_1d_0$ respectively. The routing indicator for the ith stage is defined as $t_i{=}s_i \bigoplus d_i$, where $\bigoplus$ is a bit-wise exclusive-or operator. If t_i is zero the connection of the in-coming link in the switching element is horizontally connected (straight in figure 2a) to the out-going link. Otherwise, the connection is diagonal (swap in figure 2b). An illustration of the XOR algorithm is given in figure 3.

3 Simulation Modeling of a Switching Element

The smallest component in our MIN simulation consists of a 2×2 crossbar switching element. The functionalities of each switching element include (i) receive messages from in-coming links, (ii) execute routing algorithm and (iii) forward messages to the desired out-going links. Therefore, a switching element can be treated as a queuing system. The proposed process-oriented parallel simulation model (see figure 4) consists of the following five logical processes: two packet generators, one service station

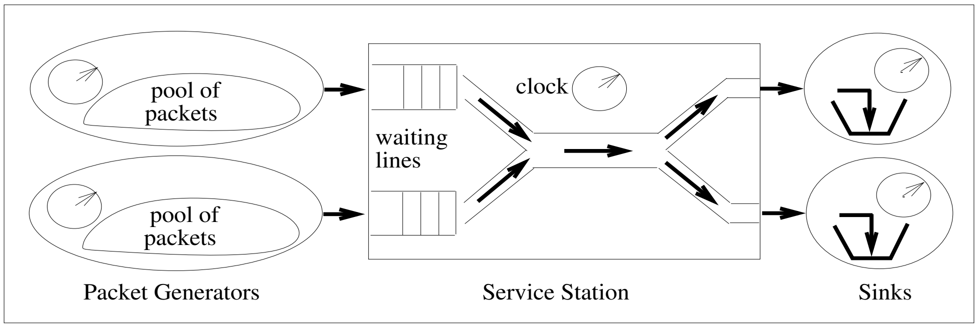


Figure 4: A Queuing Model for a 2×2 Switching Element

with parallel servers, and two sinks. Two waiting lines (or buffers) of *finite queue length* are embedded in the station to model a blocking switch. Each process has its own local clock to indicate its simulation progress. All processes used for simulation can be executed in parallel. The generator and server processes are governed by inter-message arrival time and switching time respectively, and modeled using statistical distributions. However, to model closely the operations of the switching element the arrival of a message will have to be delayed if the waiting line at the receiving link is completely filled. Therefore, a blocked message will have to wait in the generator until it is allowed to proceed, instead of being discarded as commonly assumed in mathematical analysis [13, 16]. For simplicity, we assume that the processing time required in a sink process is instantaneous as compared to the switching time. It follows that messages departing from a switching element are not blocked. We assume a constant switch delay in our simulation.

3.1 Modeling Traffic Contention

Traffic contention problem can occur in MIN's operation if the buffer associated with an in-coming link is full when a message arrives. The re-transmission approach to resolving this problem is costly because each transmission has to be acknowledged. This can be modeled using a reserved-buffer to store the transmitted messages together with a timed-out watch-dog process to decide on the execution of re-transmission. Our proposed *delayed-transmission approach*, on the other hand, incurs lesser amount of overheads, and ensures that each message will be processed by its receiver. This approach also models closely the operation of a *blocking switch*.

Some transmission protocols are required to implement the delayed transmission. In the following discussions, when a transmission contains no information other than serving as a request or reply of link status, we refer to it as a *signal*. Otherwise we refer to it as a *message*. The underlined indicates that a signal/message needs to be time-stamped.

- *Request Progress* (REQ) - This signal is used by a generator to find out if the message buffer of the service station can accommodate additional message.
- *Access Request* (ACC) - This message contains the source and destination addresses of an access request. It is sent to the service station upon receiving the reply signal "Slot Available".
- *Slot Available* (AVA) - This signal is replied by a service station to notify that the message buffer of the desired in-coming link is not full.
- *Slot Not Available* (NAV) - This signal is replied by a service station to notify that the message buffer of the desired in-coming link is full.
- *Transmit Time* (TIM) - This message contains the time for a generator to re-transmit a blocked message. It is sent by a service station when a message is removed from its message buffer where a NAV signal was previously replied to the generator on that link.
- *Stop* (STO) - This signal is used by a service station/sink to inform the generator/service station that its process is going to terminate.

To ensure that messages are not lost at a service station when the buffer is full, the simulation processes must adhere to the following transmission protocols, henceforth referred to as *hand-shaking protocol*, during transmission.

1. Before a message transmission is initiated, the sender must find out if the receiving buffer contains at least one unfilled slot. This is done by sending a REQ signal and waiting for the reply.
2. When a service station receives a REQ signal, it checks the buffer of the in-coming link. If the buffer is not completely filled, an AVA signal is returned to the sender. Otherwise a NAV signal is sent.
3. A service station has to wait for an ACC message after it replied an AVA signal.
4. When a message is removed from a buffer of a service station and a NAV signal was previously sent to the generator on that link, a TIM message containing the current clock time has to be sent to the generator.

5. A service station should wait for a REQ signal from the same link where it has sent a TIM message.
6. A generator may send out an ACC message only after an AVA signal is received.
7. A generator receiving a NAV signal has to wait for a TIM message and then re-transmit a REQ signal at the time specified in TIM message.

An illustration of how the proposed delayed-transmission protocols resolve the traffic contention problem is given in figure 5.

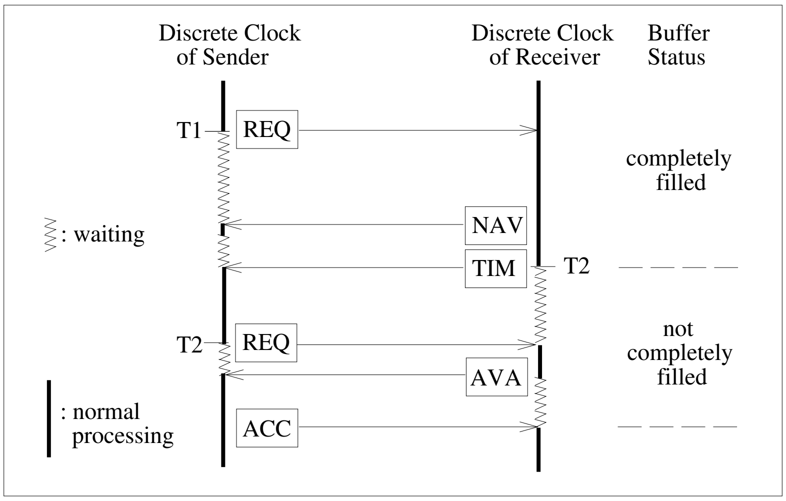


Figure 5: Resolving Traffic Contention Using Delayed-Transmission

3.2 Modeling Routing Conflict

Routing conflict occurs in a switching element when two messages coming from different in-coming links are sent to the same out-going link at the same or overlapped time interval. When this occurs, an arbitration scheme is required to serialize the accesses. In our simulation, the upper link is given the priority to proceed first. To facilitate the implementation of the priority scheme, the operations in the service station are represented by discrete events shown in figure 6.

Altogether four types of event are modeled in the simulation. The first two event types, denoted by $ARRIVAL[i]$, where i is 0/1 for upper/lower in-coming link, are the arrival events representing the processing of in-coming messages from the upper and lower links respectively (see figure 6a). The last two event type, denoted by $DEPARTURE[j]$, where j is 0/1 for the upper/lower out-going link, are the departure events representing the forwarding of messages to the jth out-going link (see figure 6b). All events are stamped with an occurrence time. It is important to note that event execution must be performed in ascending order of occurrence time. Otherwise incorrect simulation results will be produced [20]. As shown in figure 6a, the execution of an arrival event when the desired out-going link status is idle results in a change of the link status to busy followed by scheduling a departure event for the arrival. If the link status is already busy, however, the arrival event will only increment the queue length of the in-coming link. These operations mimic a real switch where a message is processed straightaway on its arrival or placed in the link buffer depending on the out-going link status. As the arrival is no longer outstanding after it is executed, scheduling the next arrival on the same link is necessary to prevent causality error [20]. Nonetheless, the immediate scheduling of the next arrival will be impossible if the message buffer is full and will have to be delayed until the execution of a departure event, making room to accommodate a new arrival. Similarly, the execution of a departure event will have to schedule the next departure if there is message(s) waiting to depart from the out-going link. In addition to simulating a finite-size buffer, a departure event will also have to schedule an arrival to the link where a previous message was blocked due to the unavailability of buffer space (refer to figure 6b).

- ARRAY[j], j = 0 to 1, are FIFO data structure where each slot in ARRAY[j] contains the source and destination of a message, and the in–coming link number.
- QUERY is a function that returns the in–coming link number of the first message in ARRAY[j]. The first message remains in ARRAY[j] after the function call.
- i/j is an index to in–coming/out–going links.
- XOR is a function that returns the out–going link number based on Exclusive–OR tag algorithm.

```
increment number_of_arrivals[i]  by 1;
j = XOR(source, dest);
if link[j] is idle
   begin
      set link[j] to busy;
      schedule a DEPARTURE[j] event
        at (clock + switch_delay);
   end
else
   increment queue_length[i] by 1;

enqueue arrival to ARRAY[j];
wait for REQ from link[i];
if (queue_length[i] < capacity)
  begin
    send AVA on link[i];
    wait for ACC from link[i];
    schedule next ARRIVAL[i] event
       at ACC's timestamp;
  end
else
  begin
    set the occurrence time of
       ARRIVAL[i] to +∞;
    set last_request_time[i] to
       REQ's timestamp;
    set link[i] to waiting;
  end;
```

(a) Arrival Event

```
dequeue a message from ARRAY[j];
set source and destination in ACC;
set ACC to current clock time;
send ACC on link[j];
if (ARRAY[j]) is empty
   begin
      set link[j] to idle;
      set the occurrence time of DEPARTURE[j]
         event to +∞;
   end
else
   begin
      schedule next DEPARTURE[j] event
         at (clocktime + switch_delay);
      i = QUERY(ARRAY[j]);
      decrement queue_length[i]  by 1;

      if link[i] is waiting
        begin
          set link[i] to not_waiting;
          if (last_request_time[i] <  clock)
            begin
              reply NAV on link[i];
              set clocktime in TIM;
              send TIM on linkl[i];
              wait for REQ from link[i];
            end;
          reply AVA on link[i];
          wait for ACC from linkl[i];
          schedule next ARRIVAL[i] event
              at ACC's timestamp;
        end
   end;
```

(b) Departure Event

Figure 6: Discrete Events for the Simulation of a Switching Element

The main control to execute the events is iterative (see figure 7). It repeatedly selects an event with the smallest occurrence time to execute. The execution of each event will update the system state,

```
set clocktime to 0;
set out–going link[0] and out–going link[1] to idle;
set queue_length[0] and queue_length[1] to 0;

wait for REQ from in–coming link[0];
reply AVA on in–coming link[0];
wait for ACC from in–coming link[0];
schedule ARRIVAL[0] event at ACC's timestamp;

wait for REQ from in–coming link[1];
reply AVA on in–coming linkl[1];
wait for ACC from in–coming link[1];
schedule ARRIVAL[1] event at ACC's timestamp;

set occurrence times of DEPARTURE[0] and DEPARTURE[1] events to +∞;

repeat
   select next event with smallest occurrence time from outstanding event(s);
   update the system state in the time interval [clocktime, next_event_time];
   advance clocktime to next_event_time;
   execute the next event according to its event type;
until clocktime >= duration_of_simulation;
```

Figure 7: Main Control for Parallel Discrete-Event Simulation

consisting of queue lengths, link status, number of arrivals and so on, and possibly schedule one or more new event(s). The clock in the service station advances together with the occurrence time of the selected event and the iteration terminates when the clock reaches the duration of simulation.

Figure 8 depicts how routing conflict problem is resolved. For illustration purpose, let the first arrival times on both links be 4 (μsec, say) and both arrivals are to be forwarded to the upper out-going link. Let the second arrival times on both links be 12 μsec and 11 μsec respectively and the switch delay be 3 μsec. The discrete-event simulation algorithm serializes the departures of two concurrent arrivals at *clocktimes* 7 μsec and 10 μsec respectively on the upper out-going link. For completeness of description, we also include the pseudo-codes for the generator and sink processes in figures 9.

Iteration Number	0	0	1	2	3	4
Discrete Clock	[0]	[0]	[4]	[4]	**[7]**	**[10]**
EOT	[0,0,+∞,+∞]	[4,4,+∞,+∞]	[12,4,7,+∞]	[12,11,7,+∞]	[12,11,10,+∞]	[12,11,+∞,+∞]
Remarks	initial values	receive ACCs of the same timestamp and same destination from both in−coming links.	ARRIVAL[0] event is selected for execution based on priority scheme. schedule next ARRIVAL[0] & DEPARTURE[0] events at clocktimes 12 & 7 respectively.	ARRIVAL[1] event is selected for execution based on minimum event occurrence time. schedule next ARRIVAL[1] event at clocktime 11.	DEPARTURE[0] event is selected for execution. send message on out−going link[0]. schedule next DEPARTURE[0] event at clocktime 10.	DEPARTURE[0] event is selected for execution. send message on out−going linkl[0].

EOT : [a, b, c, d,], where a = Event Occurrence Time of ARRIVAL[0],
b = Event Occurrence Time of ARRIVAL[1],
c = Event Occurrence Time of DEPARTURE[0],
d = Event Occurrence Time of DEPARTURE[1].

Figure 8: Serializing Routing Conflict

```
clocktime = inter_arrival_time;
while (clocktime <= duration_of_simulation)
begin
   timestamp and transmit REQ to server;
   wait for reply;
   if reply is AVA
     begin
       set source and destination addresses
          in ACC;
       timestamp and send ACC to server;
     end
   else
     if reply is NAV
       begin
         wait for TIM;
         clocktime = TIM's re−transmit time;
         timstamp and send  REQ to server;
         wait for AVA;
         set source and destination addresses
            in ACC;
         timestamp and send ACC to server;
       end
     else  { reply signal is STO }
       begin
         set server to terminated;
         set clocktime to +∞;
       end;
    increment clocktime by inter_arrival_time;
end;
```

(a) Generator

REQ : Request Progress
AVA : Buffer Available
NAV : Buffer Not Available
TIM : Transmit Time
ACC : Access Request
STO : Stop

```
arrival_count = 0;
wait for ACC;
clocktime = ACC's timestamp;
while (clocktime <= duration_of_simulation)
begin
   increment arrival_count;
   wait for ACC;
   clocktime = ACC's timestamp;
end;

send STO signal to server;
```

(b) Sink

Figure 9: Generator and Sink Processes

3.3 Modeling Parallel Packet Transmissions

In MIN's operation, parallel transmission initiated from a switching element is allowed if both out-going links are ready at the transmit time. Therefore, there would be no routing conflict if the concurrent arrivals in figure 8 were sent to different out-going links. Since the simulation model consists of two parallel servers and the assignments of departure times on the two out-going links are independent, parallel transmission in the real switch can be modeled and simulated. An illustration of how parallel transmissions are simulated is depicted in figure 10. We assume that the arrival times are the same as those in figure 8, but the concurrent arrivals at *clocktime* 4 are sent to different out-going links. Figure 10 shows that the concurrent messages are forwarded simultaneously, in terms of the discrete clock times, to their desired out-going links by the departure-event executed on different out-going links.

Iteration Number	0	0	1	2	3	4
Discrete Clock	[0]	[0]	[4]	[4]	**[7]**	**[7]**
EOT	[0,0,+∞,+∞]	[4,4,+∞,+∞]	[12,4,7,+∞]	[12,11,7,7]	[12,11,+∞,7]	[12,11,+∞,+∞]
Remarks	initial values	receive ACCs of the same timestamp & different destinations from both in–coming links.	ARRIVAL[0] event is selected for execution based on priority scheme. schedule next ARRIVAL[0] & DEPARTURE[0] events at clocktimes 12 & 7 respectively.	ARRIVAL[1] event is select–ed for execution based on mini–mum event occurrence time. schedule next ARRIVAL[1] & DEPARTURE[1] events at clock–times 11 & 7 respectively.	DEPARTURE[0] event is selected for execution. send message to out–going link[0]. set out–going link[0] to idle.	DEPARTURE[1] event is selected for execution. send message to out–going link[1]. set out–going link[1] to idle.

Figure 10: Parallel Transmissions on Different Out-going Links

Broadcast message is also a form of parallel transmissions except that it uses only one input message to generates more than one (in our case, two) output messages. The processing of broadcast messages in our simulation is slightly different from the usual processing of messages containing only one destination. If both out-going links are idle, the arrival event for the broadcast message will schedule two

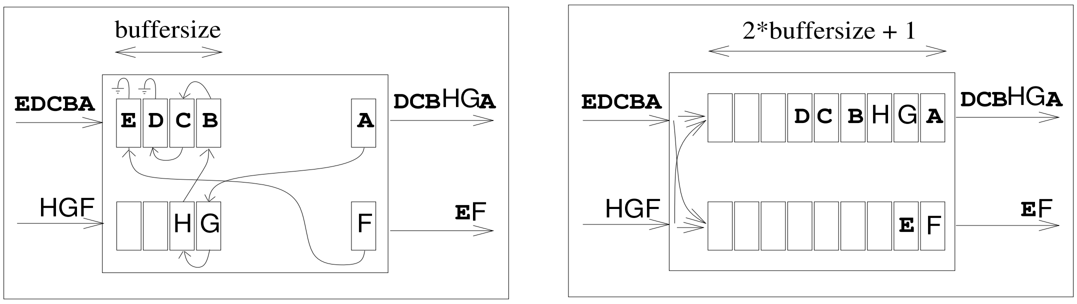
Iteration Number	0	0	1	2	3
Discrete Clock	[0]	[0]	[4]	**[7]**	**[7]**
EOT	[0,0,+∞,+∞]	[4,11,+∞,+∞]	[12,11,7,7]	[12,11,+∞,7]	[12,11,+∞,+∞]
Remarks	initial values	receive broad–cast ACC of timestamp 4 from link[0]. receive ACC message of timestamp 11 from link[1].	ARRIVAL[0] event is selected for execution. schedule next ARRIVAL[0], DEPARTURE[0], DEPARTURE[1] events at clocktimes 12, 4 & 4 respectively.	DEPARTURE[0] event is selected for execution. send message on out–going link[0]. set out–going link[0] to idle.	DEPARTURE[1] event is selected for execution. send message on out–going link[1]. set out–going link[1] to idle.

Figure 11: Simulating Broadcast Message

departure events on both out-going links of the same occurrence time (*arrival time* + *switch delay*). It is noteworthy that in the real operations the transmissions of a broadcast message may also have to be serialized or delayed if the out-going link(s) is busy. Such scenarios are also modeled in our simulation. An illustration of an upper broadcast is depicted in figure 11. Assume that the broadcast message arrives at *clocktime* 4. Let the arrival time of the second arrival on incoming link[0] and the first arrival on the in-coming link[1] be 12 and 11 respectively. Figure 11 shows that the broadcast message is forwarded on both out-going links simultaneously at *clocktime* 7.

3.4 Modeling Preemptive Packet Transmissions

Preemptive packet transmission refers to the instance where packet(s) is forwarded to its desired out-going link before the departure(s) of packet(s) arriving earlier on the same in-coming channel. Preemptive transmission is allowed in MIN when the desired out-going link of a packet becomes available while it is waiting in a buffer. Consider five arrivals A, B, C, D, E of timestamps 1, 5, 6, 7 and 8 respectively on in-coming link[0], where the first four arrivals are sent to out-going link[0], and the fifth arrival to out-going link[1] (see figure 12). Let the timestamps of the three arrivals on in-coming link[1], denoted by F, G, H be 2, 3 and 4 respectively, and the first arrival is sent to out-going link[1] and the second and third arrivals to link[0]. Let the switch delay be 10. If the in-coming messages on link[0] are processed in sequential order, E will not be processed before D is removed from the message buffer at *clocktime* 51 $(1 + 5 \times 10)$. In fact, E can be processed earlier because its desired out-going link is available after F departs the switching element at *clocktime* 12 $(2 + 10)$. Therefore, the $DEPARTURE[1]$ event executed at *clocktime* 12 (for F) may schedule the next $DEPARTURE[1]$ event at *clocktime* 22 $(12 + 10)$ (for E). To model the preemptive packet transmission, the in-coming messages are partitioned into two groups for out-going link[0] and link[1] respectively on their arrivals. Each group of messages is then sorted in ascending arrival-timestamp order. Pointers (figure 12a) or circular arrays (figure 12b) may be used for the data structure. In our implementation circular arrays are used to model the preemptive packet transmissions.



(a) Pointers (b) Circular Arrays

Figure 12: Data Structures for Preemptive Packet Transmissions

3.5 Resolving Forward-Cascading Deadlock Using Pseudo-Arrival

Although the discrete-event simulation model using the proposed hand-shaking protocols is able to produce correct simulation results, deadlock may occur in the service-station when the simulation approaches the completion time. The reason is that the execution of an arrival event will have to schedule the next arrival. However, if the transmit time of the desired ACC message is beyond the completion time the generator will not activate the transmission, resulting in deadlock in the process on its receiving end (see figure 13a). In consequence, deadlock is cascaded forward to the sink process. An approach to resolve this problem is proposed in [20]. The generator and service-station processes intentionally send out a pseudo-arrival consisting of REQ signal and an ACC message both with the timestamp of $\tau + \delta$, where τ is the duration of simulation and $\delta > 0$, on all their out-going links when their clock reaches the completion time (see figure 13b). Note that in this instance no reply is required after the REQ signal is sent because the receiving process may have been terminated, thereby causing deadlock in the sending process if it waits for a reply signal.

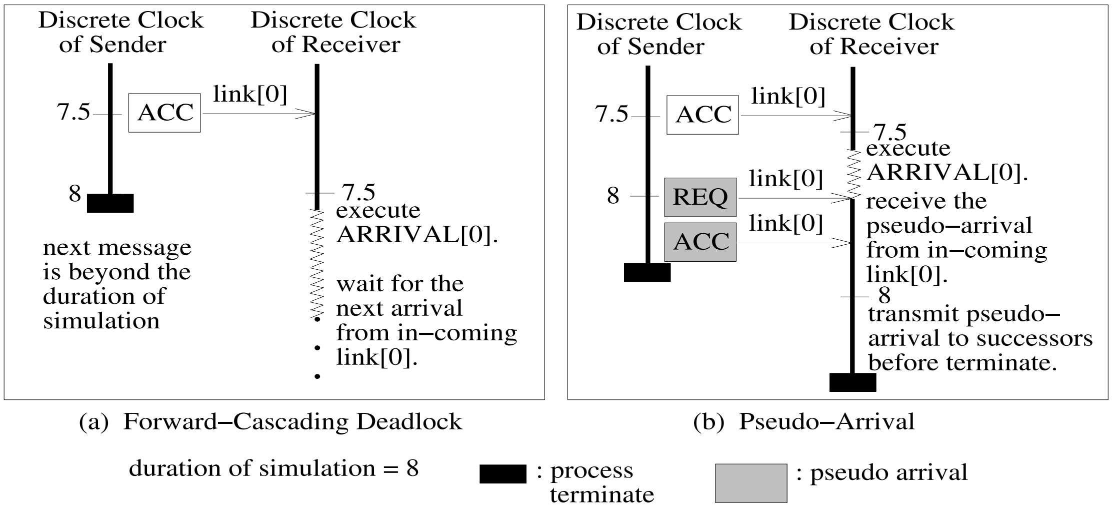


Figure 13: Solving Forward-Cascading Deadlock Using Pseudo-Arrival

3.6 Resolving Backward-Cascading Deadlock Using STO Signal

Even with the inclusion of the pseudo-arrivals, deadlock may still occur in the parallel simulation. Consider a service station where the generator on link[0] is at the wait state due to unavailability of buffer space. Therefore the $ARRIVAL[0]$ event in the receiving process will not be selected for execution until a departure event that creates space on link[0] is executed. Suppose the occurrence time of $DEPARTURE[0]$ and $DEPARTURE[1]$ are also beyond the duration of simulation. If the receiving process executes an $ARRIVAL[1]$ event and schedules the next $ARRIVAL[1]$ such that the occurrence time is beyond the completion time, the receiving process will be terminated in the next iteration because all the four events are not eligible for execution. Consequently, the generator process on link[0] will be locked indefinitely (see figure 14a). It is clear that deadlock will be cascaded backward

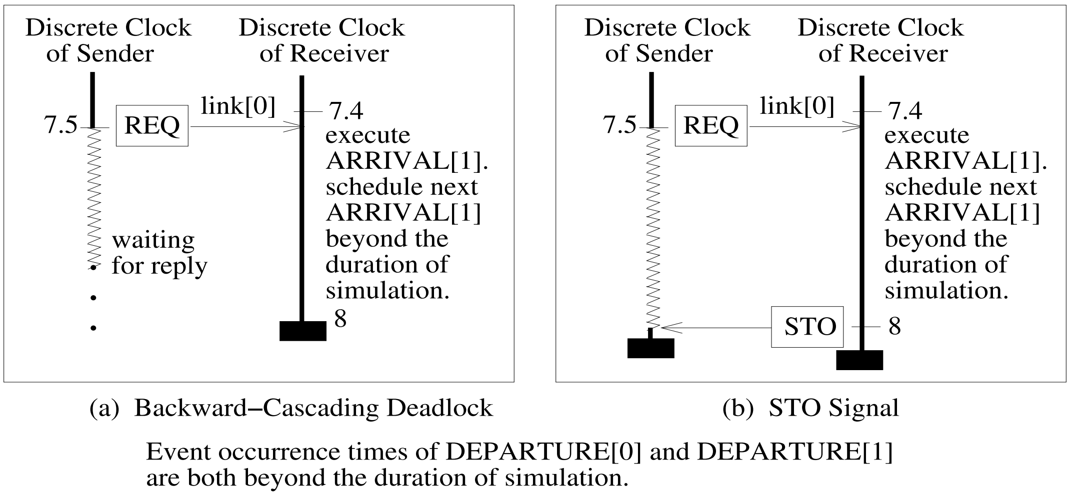


Figure 14: Resolving Backward-Cascading Deadlock Using STO Signal

if the generator is also connected to the out-going links of some processes. This problem can be resolved by the service-station sending a STO signal to the generator before it terminates. On the other hand, after a generator sends a REQ signal it should also accept the STO signal as the reply, not just the AVA and NAV signals only. If the reply is a STO signal, meaning that its receiving process has already terminated, the generator should also terminate (see figure 14b).

4 Parallel Simulation of MIN

The process-oriented nature for modeling and simulating a switching element can be extended to simulate MIN. In the operation of a MIN all switching components operate autonomously and so does its counterparts in simulation. Each switching component in the MIN is simulated by a service-station process executing discrete events. The interactions among the processes is by passing signals and messages, and all processes are executed in parallel. A simulation configurer is used to spawn the processes for packet generators, switching elements and sinks, and to establish the switch interconnections by sending relevant process-ids to each process. Enhancements to the simulation model for the switching element discussed earlier are necessary. The departures of messages from a service station, except those in stage-0, have to adhere to the hand-shaking protocols specified in section 3.1. The reason is that the capacity of the message buffer for each receiving link is constrained and therefore messages will be lost when the receiving buffer is full. We now include additional notations necessary for the subsequent discussions. Let the occurrence times of $ARRIVAL[i]$ and $DEPARTURE[j]$, $i, j \in \{0,\ 1\}$, be $toa[i]$ and $tosc[j]$ respectively. Let $i' = i \oplus_2 1$, and $j' = j \oplus_2 1$, where $\oplus_2$ is an additive operator of modulo two.

4.1 Circular Wait Problem

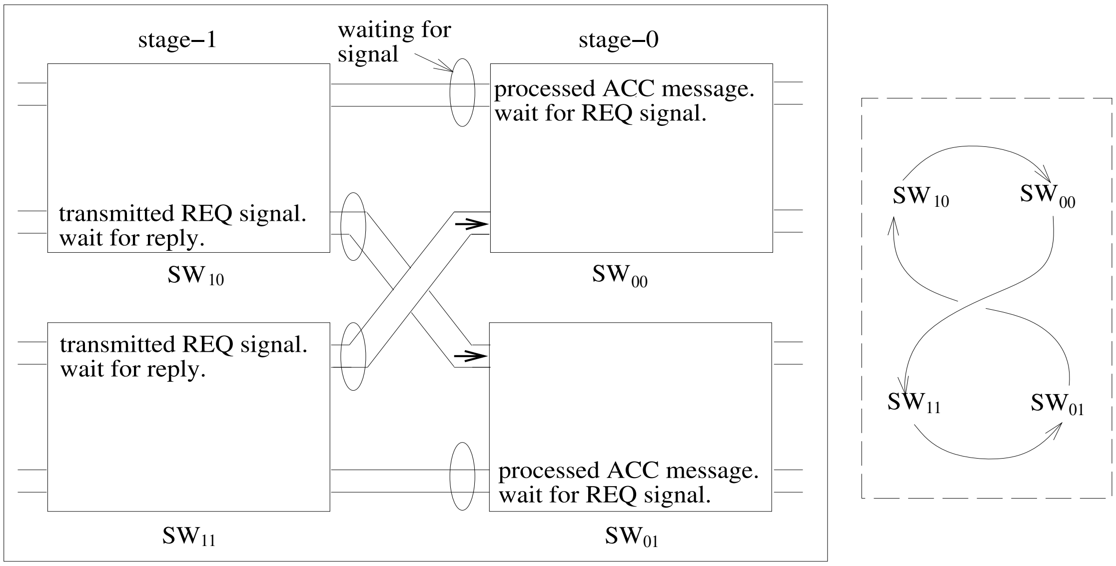
The inclusion of pseudo-arrival and termination signal (STO) in the simulation model for a switching element prevents deadlock when the simulation approaches the completion time. Such measures are sufficient to ensure a deadlock-free parallel simulation only for *minimum* MIN with *infinite* buffer space. As each transmission can be accommodated by the receiver (due to unlimited buffer storage), and *looped* interconnection does not exist among the switching elements (due to unique path property of minimum MIN), there is no danger of deadlock in the MIN simulation. In other words, the inter-process communication may be asynchronous and uni-directional, and therefore no transmission protocol is necessary. The assumption of infinite buffer is also commonly used in mathematical models of MIN to simplify the analysis in order to yield tractable solutions [16]. For finite-size message buffer, on the other hand, a dialog between the sender and receiver is required before a packet transmission is activated in order to prevent loss of messages. As each process may communicate with four other processes during simulation, there is a danger that after a process sends a REQ signal no response is received because the receiving process is not processing the event on the desired link. As the communication is bi-directional, such waiting processes may happen in a loop, resulting in infinite wait to all waiting processes. The *circular-wait deadlock* may also occur in *fault-tolerant* MIN simulations that assume infinite buffer space due to the existence of *looped interconnections*. Deadlock problem has already been well-studied and proposed solutions, such as using *null messages* are widely available in the literature [6, 14]. Nonetheless, in the MIN simulation the problem becomes worse due to bi-directional communication and increased complexity for higher dimension interconnection networks. Switching element with finite buffer is assumed in this paper.

4.2 NULL Messages Without Guaranteed-Lookahead

A modified null-message approach is used to resolve the deadlock problem in our MIN simulation. It is vital to note that in the conventional null-message approach, whenever a message of timestamp σ is transmitted on an out-going link, a null message containing a *time indicator* of $\sigma + \epsilon$, for $\epsilon > 0$, is also transmitted on all other out-going links to inform the receivers to proceed, instead of waiting, with their simulation up to the time indicated [14]. These null messages are then circulated with their indicators incremented by ϵ whenever they are forwarded from one process to the others. Unfortunately, such *lookahead* indicators are not available in MIN simulation when the message buffer(s) is completely filled. For some instances of the bi-directional multi-link communication, the time indicators of the forwarded null messages are always equal to the indicator of the received null message (see section 4.3). As a result, all processes will not schedule any event for execution after receiving the forwarded null messages. If null messages are relayed to the other processes without guaranteed lookahead, they may cause *livelock*; meaning that all processes are busy in relaying null messages without any progress in their local clocks. The way we avoid the livelock problem is by giving a higher priority to departure-event execution whenever a $tosc[j]$ is equal to a null-message indicator. By using this priority scheme in the event scheduling, a process will proceed with the simulation instead of keep relaying null messages to other processes.

4.3 Resolving Deadlock

Consider a simple 4×4 Omega network as shown in figure 15a where SW_{ij} represents the jth switching element in the ith stage. This figure corresponds to the instance where SW_{10} and SW_{11} are executing departure events, and SW_{00} and SW_{01} are executing arrival events. Both SW_{10} and SW_{11} are waiting for the status of the receiving buffer on their lower and upper out-going links respectively, while SW_{00} and SW_{01} are waiting for new arrival from their upper and lower in-coming links respectively.



(a) An Instance of Deadlock

(b) Four Processes Waiting in a Loop

Figure 15: An Instance of Circular Wait in 4×4 MIN Simulation

This results in a *deadlock* situation among the four switching elements (see figure 15b). A more complicated deadlock instance is shown in figure 16. For generality, we include the additional transmission protocols, henceforth referred to as *relaying protocols*, to resolve the deadlock problem in the MIN simulation for higher dimensions. In the following descriptions, α, β, and γ are the time indicators to inform the sender/receiver on the various links that the process will not send them any useful (hand-shaking) signals and messages before the specified time.

1. When a REQ signal of timestamp Θ is sent to an out-going link$[j]$, three NULL messages containing the time indicators α, β and γ respectively, where

$$\alpha = \begin{cases} toa[0] & \text{if in-coming link[0] buffer is not full} \\ \Theta & \text{otherwise} \end{cases}$$

$$\beta = \begin{cases} toa[1] & \text{if in-coming link[1] buffer is not full} \\ \Theta & \text{otherwise} \end{cases}$$

$$\gamma = \begin{cases} tosc[j'] & \text{if out-going link}[j'] \text{ has a message} \\ & \text{scheduled for transmission} \\ \Theta + switch\ delay & \text{otherwise} \end{cases}$$

are also sent to in-coming link[0], in-coming link[1] and out-going link[j'] respectively.

2. When a NULL message is received while a process is waiting on an out-going link[j] for the reply to a REQ signal, the process should also forward the NULL message on the other three links with the time indicators specified in clause 1.

3. When scheduling a new arrival event on an in-coming link[i], a process should wait for a REQ signal and also process a NULL message if it arrives.

4. If a NULL message is received in clause 3, three NULL messages containing the time indicators α, β and γ respectively, where

$$\alpha = \begin{cases} toa[i'] & \text{if in-coming link}[i'] \text{ buffer is not full} \\ \min\ (tosc[0],\ tosc[1]) & \text{otherwise} \end{cases}$$

$$\beta = \begin{cases} tosc[0] & \text{if out-going link}[0] \text{ has a message} \\ & \text{scheduled for transmission} \\ \Theta + switch\ delay & \text{otherwise} \end{cases}$$

$$\gamma = \begin{cases} tosc[1] & \text{if out-going link}[1] \text{ has a message} \\ & \text{scheduled for transmission} \\ \Theta + switch\ delay & \text{otherwise} \end{cases}$$

are also sent to in-coming link[i'], out-coming link[0] and out-going link[1] respectively.

With reference to clause 1, a process executing departure event corresponding to link[j] will not response with hand-shaking signal/message, to link[i], $i = 0$ to 1, before executing the next $ARRIVAL[i]$ event. If the $ARRIVAL[i]$ event has not been scheduled (due to buffer full) for execution, the earliest response time is computed by assuming that the *current* departure will make room in link[i] message buffer so that the desired arrival can be scheduled *immediately*. The response time of link[j'] is the occurrence time of the $DEPARTURE[j']$ event if it is already scheduled. Otherwise the earliest response time is computed by assuming that the current arrival will depart the process through link[j'] after a switch delay interval.

With reference to clause 3, if a null message of indicator, denoted by χ, is received while scheduling a new arrival, the receiving process should continue with its simulation without considering the $ARRIVAL[i]$ event. Simulation proceeds as per normal when the clock time in the null-message receiving process reaches χ, i.e., it should schedule a new $ARRIVAL[i]$ event when its clock reaches the indicator.

With reference to clause 4, a process executing an arrival event corresponding to link[i] will not response to link[i'] before executing the next $ARRIVAL[i']$ event. If the $ARRIVAL[i']$ event has not been scheduled (due to buffer full) for execution, the earliest response time is computed by assuming that the *next* departure event execution will make room in link[i'] message buffer so that the desired arrival can be scheduled immediately after the departure. The response time to link[j], $j = 0$ to 1, is the event occurrence time of $DEPARTURE[j]$ event if it is already scheduled. Otherwise the earliest response time is computed by assuming that the current arrival will depart through out-going link[j] after a switch delay interval.

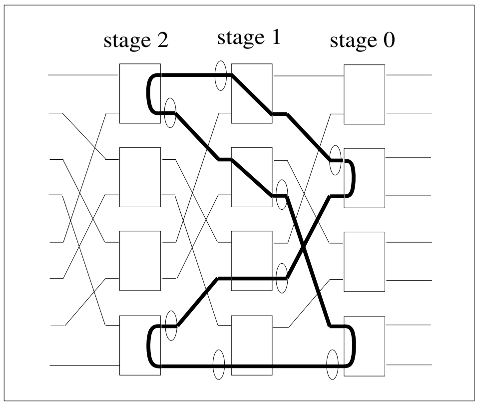


Figure 16: An Instance of Circular-Wait in a 8×8 MIN Simulation

4.4 Reducing Null-Message Overhead

The transmission protocols, including hand-shaking and relaying, are sufficient to ensure that deadlock will not occur during the conservative simulation. However, the amount of null messages generated grows rapidly, even at the beginning of the simulation run. As the transmission of each REQ signal will generate three null messages and the receipt of each null message will further generate another three null messages, it turns out that the number of null messages at the execution time t is $3^{f(t)}$, where f is a monotonic increasing function. Our experiments show that test runs can be completed only for less than 25 events. For other test runs with larger number of events, such a combinatoric explosion will abort the execution of processes, even at the initial stage of the test runs. The abortion is caused by insufficient memory for message buffer, apparently due to the flooding of null messages. Reducing null-message overhead has been studied by Cai and Turner [3]. They used carrier-null method where the routing history is appended to a null message during circulation. Based on the routing history, circular dependency of simulation can be detected and some preemptive process executions can be performed to resolve deadlock. As the carrier-null method assumes that lookahead is guaranteed in null messages, it cannot be used in our simulation. We propose four mechanisms to reduce the number of null messages. The fourth mechanism is used in our MIN simulation and its performance is discussed in section 5. In the following discussions, we refer to a process waiting on in-coming link as *receiver*, and on out-going link as *sender*.

4.4.1 Reduce by Sender and Receiver

In each switching element, we maintain four null-message records which store the *maximum* of the time indicators transmitted on each link. Before sending/forwarding a NULL message, the switching element checks the null message's time indicator, denoted by ν, and the desired link's indicator record, denoted by ρ. If $\nu \leq \rho$, the null message is not transmitted. Otherwise, the record is updated by the indicator and the null message is transmitted. This mechanism effectively reduces the volume of the null messages from *exponential* to *linear*. However, for some instances when the null-message records of the processes waiting in a loop are of the same value, *link silence* (deadlock) will happen in the simulation.

4.4.2 Reduce by Receiver Only

The null-message records are still required and updated as described previously, but trimming is performed by receivers only. The sender while executing a departure event will still send three null messages to the other three links when it sends a REQ signal or receives a NULL message. This mechanism does not introduce the link-silence problem described earlier because the sender always relays the null messages. However, the amount of null messages remains as exponential. Assume the best case where one of the three null messages sent by a sender is always absorbed by a receiver. It turns out that the number of null messages at the execution time t is $2^{f(t)}$.

4.4.3 Reduce by Sender Only

This mechanism is similar to the previous description and the volume of null messages at the execution time t is $2^{f(t)}$ in the best case.

4.4.4 Flushing In-Coming Null Messages

In this mechanism null-message records are not required. Each sender and receiver sends the null message as described in the relaying protocols. When a null message is received by a process, the sender/receiver will flush in all the null messages that have already arrived on that link and select the null message with the *largest* time indicator. The forwarding of null messages is performed as described in the relaying protocols but only on the largest time indicator and the rest are discarded. Our implementation results show that the amount of null message overhead is effectively reduced from *exponential* to *linear* when this mechanism is used.

5 Implementation Details

We have implemented the parallel simulation algorithm in the C language. It runs on a network of workstations which mimics a distributed-memory parallel computer. Before we discuss the performance results, we briefly describe our implementation platform, and present a transformation technique to map a simulated MIN onto the processors in the network.

5.1 Implementation Platform

PVM (Parallel Virtual Machine) is a software package that links a heterogeneous network of UNIX-based parallel and serial computers to work as a single concurrent computer resource [7]. PVM provides facilities for spawning, communication, and synchronization of processes over a network of heterogeneous machines. It allows its user to create processes on its own and other computers, and to send messages. The PVM system consists of two parts. The first part is a daemon process, called *pvmd3*, which must be spawned in all computers making up the virtual machine. It's main task is to handle the message communication among processes located in the same and on different computers. The second part of the PVM system is a library of interface routines for passing messages, spawning processes, coordinating tasks and modifying the virtual machine. Before executing the simulation program, we have to configure the virtual machine. It is done by running a software utility, called *pvm*, to spawn the daemon on each workstation to be included in the network.

5.2 Mapping MIN Simulation onto Multicomputers

To simulate an $n \times n$ MIN it requires s processes, where $s = n + (\frac{n}{2} \times \log_2 n) + n$. Parallelism in the simulation can be fully exploited provided the number of processors available, denote by p, is at least equal to s. Otherwise, some workstations (processors) will have to execute more than one process.

Two aspects, load balancing and communication overhead, should be considered when mapping the simulated system to the virtual machine [20]. In the first aspect, if the destination addresses are uniformly distributed among all the output components, the workload in each service station will be even. Otherwise, skewed workload may occur in some service stations depending on the distribution. The workloads introduced by the generator and sink processes are small as compared to that of a service station as they do not have to simulate the switch operations. Therefore, in the simulation run each generator process is placed together with the service-station process on its out-going link in a same processor, while each sink process is together with the service-station process on its in-coming link. In the second aspect, although there is no clear comparative indicator on intra-processor and inter-processor communication overheads, it is clear that the communication overhead incurred from the first type is less than the second type based on locality consideration. When mapping the simulation program onto the virtual machine it is therefore essential to minimize the overheads due to inter-processor communication, on the condition that the increase in intra-processor communication overhead is comparatively insignificant as compared to the decrease in that for inter-processor communication. This objective can be achieved by applying transformation technique to transform the interconnections to a more modular equivalent network, and yet still preserve the access capability of the original configuration. The following describes three schemes for mapping a 16×16 Omega network onto eight processors: *horizontal partitioning* (see figures 17), *vertical partitioning* (see figure 18), and *modular partitioning* (see figure 19). Each rectangular block can be mapped onto one processor. The first two schemes are self-explanatory.

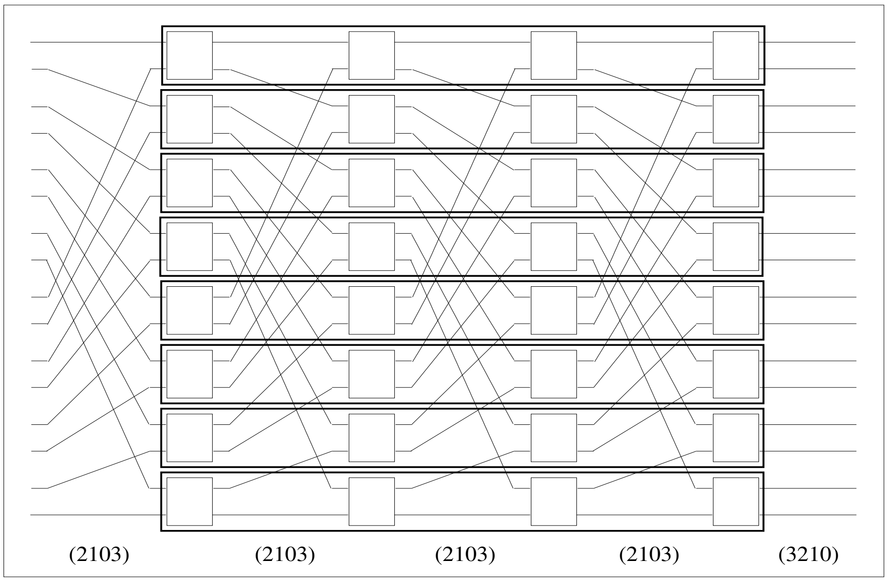


Figure 17: A Horizontal Partitioning Scheme

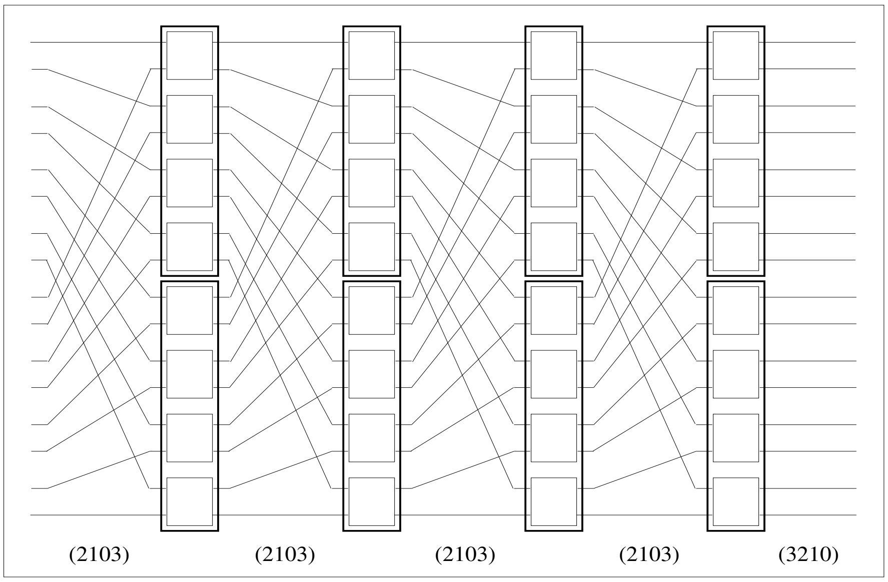


Figure 18: A Vertical Partitioning Scheme

The interconnection network shown in figure 19 is partitioned into eight smaller 4×4 Omega networks of translation number (3201). To reduced the inter-processor communication overhead, each partition is mapped onto one processor. Figure 19 is equivalent to the 16×16 Omega switch shown in figures 17 and 18. Before deriving the modular connections, we briefly highlight two properties of interconnection networks.

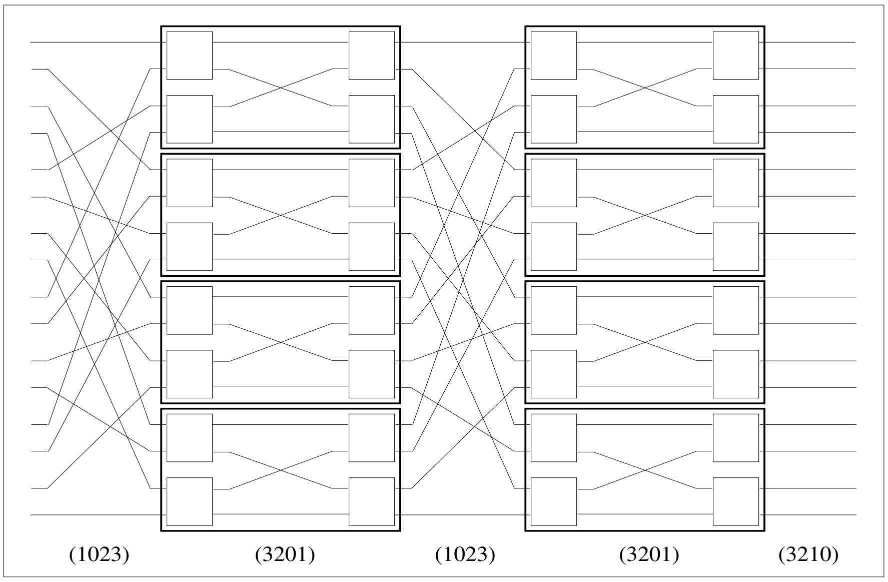
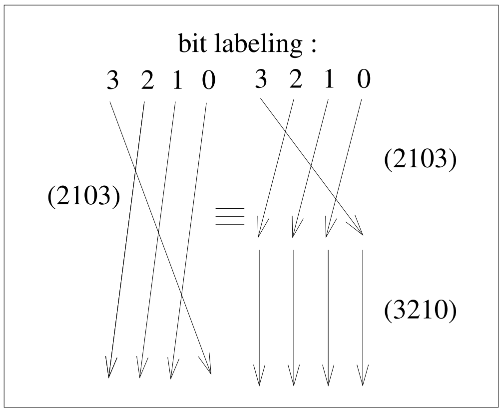
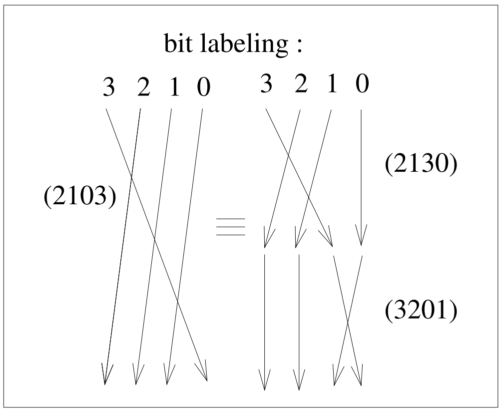


Figure 19: A Modular Partitioning Scheme

- *A connection can be decomposed into two sets of connection adjacent to each other with no change in access capability.* A trivial example is appending the identity connection $(.., i+1, i, .., 1, 0)$ to an existing connection (figure 20a). Figure 20b contains a non-trivial illustration.



(a) A Trival Decomposition
(2103) = (2103)(3210)



(b) A Non–Trival Decomposition
(2103) = (2130)(3201)

Figure 20: Decomposing an Interconnection

- *Any two adjacent connections such that one of the translation number representation is of the form* $(.., b, a, 0)$ can be combined into one connection with no change in access capability.

Based on these two properties, we transform the Omega network for modular partitioning as follows.

(2103) *(2103)* (2103) *(2103)* (3210)

=(2103) *(2130)(3201)* (2103) *(2130)(3201)* (3210)

=*(2103)(2130)* (3201) *(2103)(2130)* (3201)(3210)

=(1023)(3201)(1023)(3201)(3210)

In the above transformation, we first decompose the second and the fourth translation numbers into (2130) and the modular connection (3201). Next, we combine (2130) with (2103) but reserve the modular connection, yielding the desired configuration.

5.3 Performance Analysis

The performance of the conservative simulation algorithms was evaluated using *eight* workstations running PVM unless stated otherwise. Elapsed time and CPU time are collected. Due the the multi-user and multi-tasking operating environment, and the fact that workstations constituting the virtual machine are sited in different locations, the communication overheads are high. In addition, our workstations are also of different clock rates. Therefore, in the following analysis the performance of the parallel simulation will be gauged by its trends, instead of the individual timings. Exponential distribution is used to generate the interarrival time, and a constant is used for the switch delay. Two sets of experiment, infinite buffer size and finite buffer size, are conducted using the horizontal (HPS), vertical (VPS), modular (MPS) and balance (BPS) partitioning schemes. The balance partitioning is performed by the PVM system, based on the machine performance and machine load when processes are spawned, and we have no control on the process grouping if this option is chosen. In the following comparisons η represents the number of packets sent per generator. Due to the huge number of parameters used in MIN simulation, we give a summary of the performance results. A detailed experimental analysis will be presented in a separate paper.

5.3.1 Infinite Buffer Size

In this set of experiments we exclude the hand-shaking and relaying protocols from the program as there is no danger of deadlock during simulation. The transmission of messages is asynchronous and uni-directional. Figure 21 shows that the elapsed time of HPS is better than MPS. Our explanation to

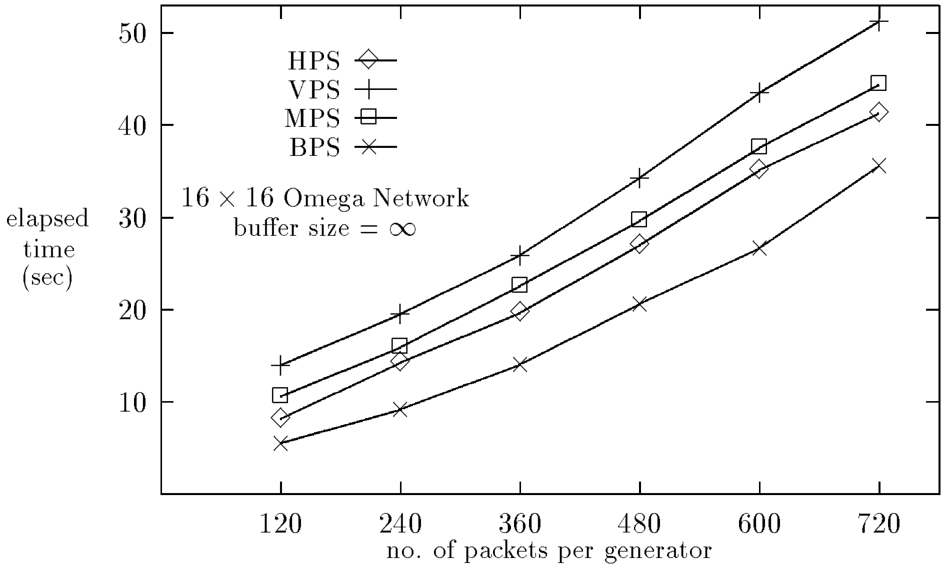


Figure 21: Elapsed Time for Different Partitioning Schemes (Infinite Buffer Size)

such a behavior is that for a fixed number of processors, say $p = 8$, in the HPS the input links of each processor are originated from at most two other processors, while in the MPS are at most four (refer to figures 22a and 22c). As the processing of the messages is serialized at a receiving processor, it turns out that the degree of parallelism in HPS is higher than MPS, yielding a smaller elapsed time in HPS as compared to MPS. The BPS out-performs the other three schemes because the load-balancing factors are taken into consideration at run time by the PVM system when spawning new simulation processes. Therefore all processes progress at close pace, reducing the communication overhead throughout the simulation. The elapsed time of VPS is the worst due to uneven distribution of generator and sink processes on the workstations.

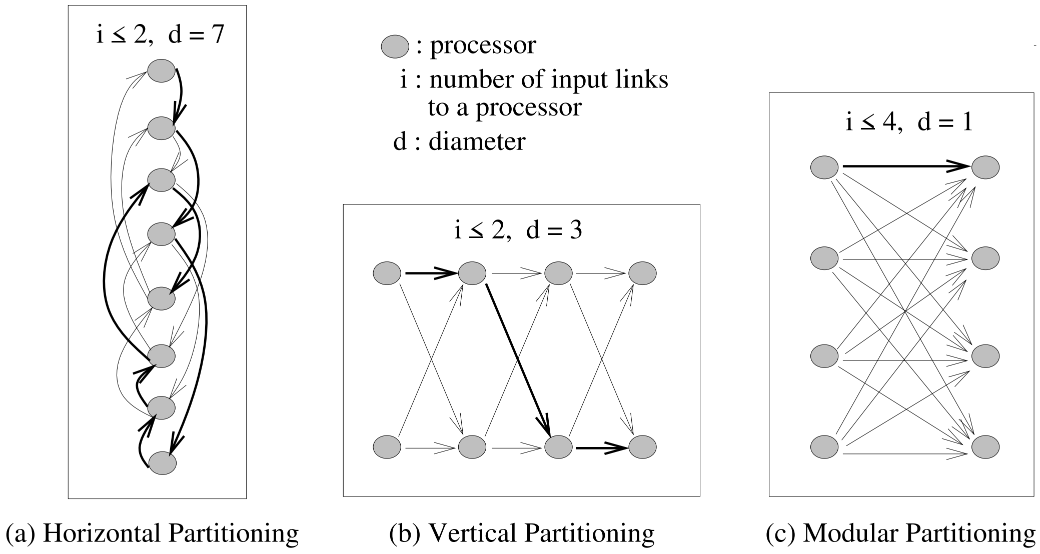


Figure 22: Inter-Processor Communication for Eight Workstations

The variations of elapsed time and CPU time versus the number of processors are depicted in figures 23 and 24 respectively. As shown in the figures, both elapsed time and CPU time of all partitioning schemes decrease as more processors are used. The variations of elapsed time and CPU time for the four mapping schemes follow the discussions in the previous paragraph.

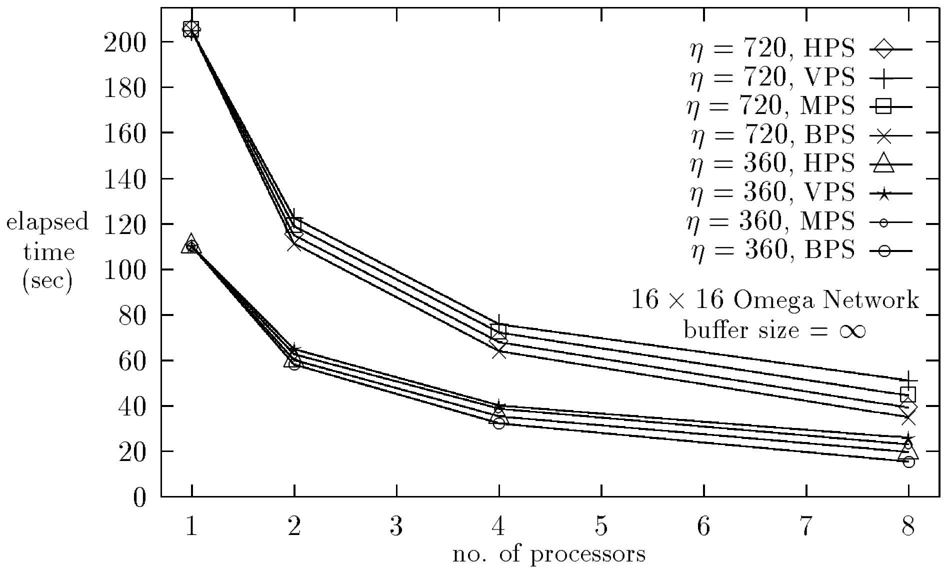


Figure 23: Elapsed Time versus No. of Processors (Infinite Buffer Size)

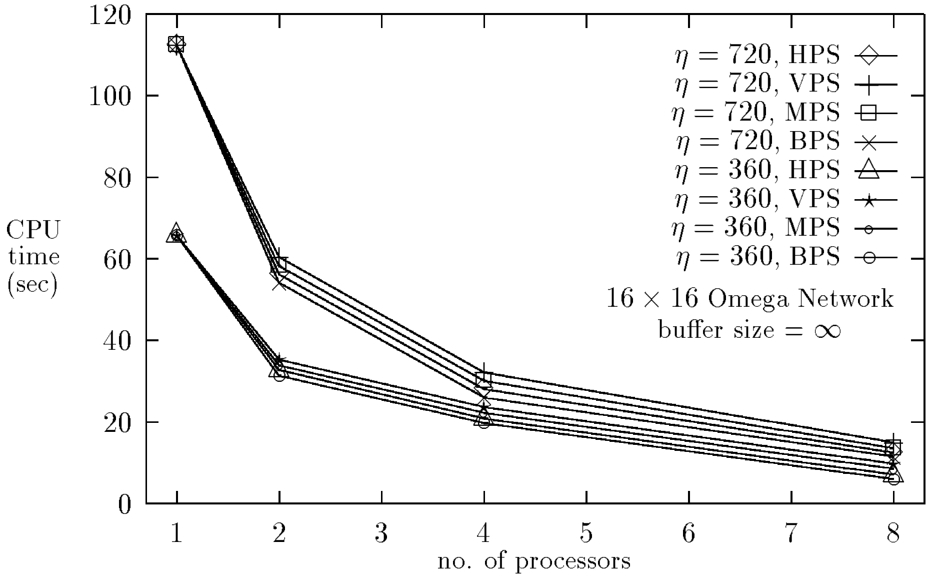


Figure 24: CPU Time versus No. of Processors (Infinite Buffer Size)

5.3.2 Finite Buffer Size

In this set of experiments we include the hand-shaking and relaying protocols in the simulation and buffer size is set to four. The transmission of signals and messages is synchronous and bi-directional. Figure 25 shows that the elapsed time of HPS, contrary to the timings collected for infinite buffer size, is larger than that of MPS. Note that for synchronous transmission, the waiting for arrival in a process causes starvation to all its succeeding processes. Such an effect results in increased elapsed time if the diameter, denoted by d, of the processor networking, the longest path length of the inter-processor connections (refer to figure 22), is long. Therefore, the elapsed time of HPS ($d = 7$) is larger than that of MPS ($d = 1$). The elapsed time of VPS is larger than those for HPS and MPS due to the uneven distribution of generator and sink processes on the workstations. Our investigation also shows that the diameter of processor networking produced by the PVM when BPM is used is always seven for eight workstations, meaning that the starvation effect of BPS is more critical as compared to VHS and MPS. The placement of processes using BPS is scattered on the workstations as compared to HPS. Therefore the inter-processor communication overhead for BPS is larger than that for HPS. In total, BPS incurs a larger elapsed time as compared to HPS.

It is well-known in process-oriented parallelism that good speedup can be achieved only if the grain-size of each process is coarse. Based on the performance results obtained from the infinite-buffer simulations, we assert that if the workload in each simulation process outweighs its communication overhead, the speedup of the finite buffer simulation will be close-to-linear. In a separate set of experiments using HPS to verify the assertion, the workload in each process was gradually increased to compute the speedup metric. The parameters used in the investigation were $\eta = 60$ and $p = 1, 2, 4$ and 8. Let ξ be the addition workload per event. In the investigation ξ is simulated by a nested *for loop* with runtime ranging from 0.02 to 0.1 second. Figure 26 shows that as the workload to execute an event is increased so is the speedup of the parallel simulation. Such an observation shows that a close-to-linear speedup is attainable even when transmission protocols are imposed in the conservative simulation, provided the workload in each process is high.

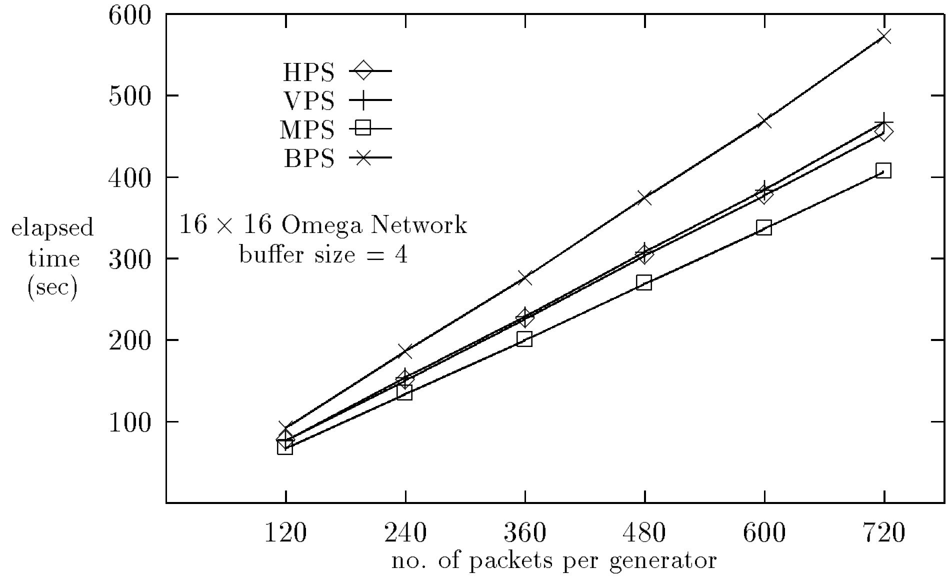


Figure 25: Elapsed Time for Different Partitioning Schemes (Finite Buffer Size)

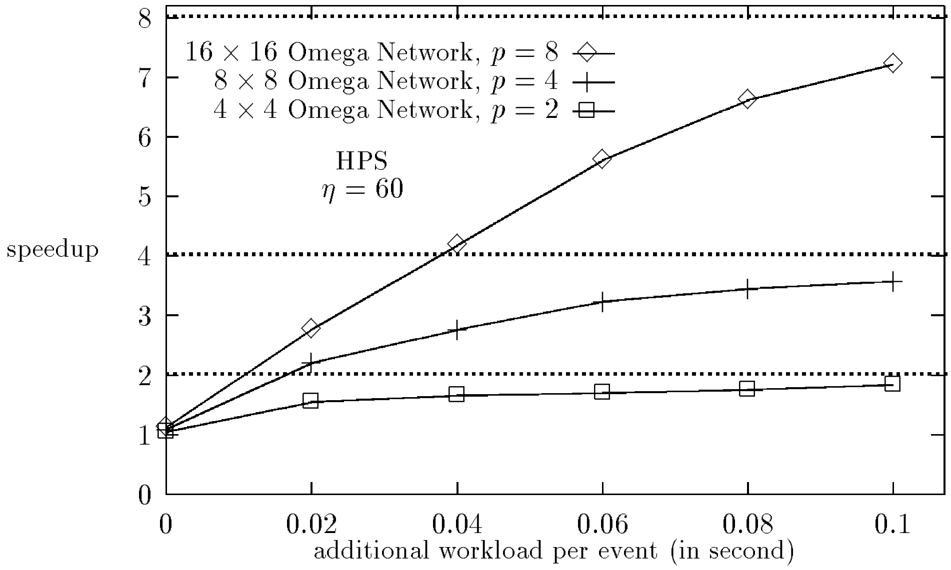


Figure 26: Speedup versus Additional Workload per Event (Finite Buffer Size)

The number of null messages, protocol signals and messages, and the number of packets (ACC messages) transmitted are also recorded. Table 1 shows the respective metrics for $\xi = 0$. The null message ratio remains fairly constant for a given partitioning scheme regardless of the number of packets transmitted. This shows that the flushing of in-coming null message is an effective mechanism to prevent the combinatoric explosion. Table 1 also sheds light on the variations in elapsed time for various partitioning schemes. As the increase in the null-message ratio correlates to the elapsed time in figure 25, we conclude that the increase in the elapsed time for various partitioning schemes is caused by the workload in processing the increasing null messages and the additional null-message communication overhead. The best partitioning scheme for the finite-buffer MIN simulation is MPS due to the close proximity of processes for intra-processor communication, and higher degree of parallelism for inter-processor communication. Although the null-message ratio for all the partitioning schemes are high there should be no alarm. Our justification to the statement is based on the complexity of the networking. In conservative simulation, transmission and relaying of null messages are required to prevent each process from waiting for itself. In the finite-buffer MIN simulation, each inter-process link is a feedback loop (bi-directional) and therefore every transmission of a useful packet is accompanied by the transmission of three null messages, resulting in *exponential* growth of null-message overhead. Nonetheless, the proposed flushing scheme reduces the overhead to *linear*.

scheme	percentage (%)	no. of messages sent per generator 120	240	360	480	600	720	average
horizontal	$\frac{Null}{Total} \times 100$	79.82	79.80	79.86	79.90	79.96	79.83	79.86
	$\frac{Protocols-ACC}{Total} \times 100$	13.46	13.41	13.40	13.38	13.31	13.44	13.40
	$\frac{ACC}{Total} \times 100$	6.72	6.79	6.74	6.72	6.73	6.73	6.74
vertical	$\frac{Null}{Total} \times 100$	80.05	79.88	79.93	79.96	80.05	79.96	79.97
	$\frac{Protocols-ACC}{Total} \times 100$	13.27	13.38	13.35	13.35	13.28	13.35	13.33
	$\frac{ACC}{Total} \times 100$	6.68	6.74	6.72	6.69	6.67	6.69	6.70
modular	$\frac{Null}{Total} \times 100$	78.60	78.71	78.97	79.12	79.05	78.96	78.90
	$\frac{Protocols-ACC}{Total} \times 100$	14.23	14.02	14.03	13.92	13.97	14.04	14.04
	$\frac{ACC}{Total} \times 100$	7.17	7.27	7.00	6.96	6.98	7.00	7.06
balance	$\frac{Null}{Total} \times 100$	81.59	81.38	82.00	81.91	81.47	81.64	81.67
	$\frac{Protocols-ACC}{Total} \times 100$	12.25	12.38	11.97	12.05	12.34	12.24	12.21
	$\frac{ACC}{Total} \times 100$	6.16	6.24	6.03	6.04	6.19	6.12	6.12

Table 1: Percentage of Signals and Messages ($\xi = 0$)

6 Conclusions

We have developed parallel simulation algorithms to analyze the performance of MIN more realistically as compared to mathematical analysis. Our work shows that the MIN simulation for infinite buffer size is straightforward as there is no danger for the occurrence of deadlock during simulation. The complexity of the MIN simulation for finite buffer size, however, increases enormously because every inter-process link is a feedback loop, and therefore additional hand-shaking protocols and relaying protocols are required to produce correct simulation results and to prevent circular-wait deadlock. In the worst case, the conservative simulation may incur large number of null-messages, aborting the simulation during execution time. We have shown that by flushing the in-coming null messages, the overhead can be effectively reduced from *exponential* to *linear*. Our investigation also shows that the placement of simulation processes on the workstations affects the elapsed time of the program. In this aspect, an efficient transformation technique, giving a modular equivalent MIN, to reduce inter-processor communication overhead for the finite buffer simulation is proposed. Our experimental results show that the speedup of conservative parallel simulation can be close-to-linear, even with the imposed transmission protocols, if the system to be simulated can be partitioned into a sufficient number of coarse-grain logical processes.

References

[1] G. S. Almasi and A. Gottlieb, *"Highly Parallel Computing"*, The Benjamin/Cummings Publishing Company, Inc., 1989.

[2] G. H. Barnes and S. F. Lundstrom, *"Design and Validation of a Connection Network for Many-Processor Multiprocessor Systems"*, IEEE Computer, pp. 31-41, 1981.

[3] W. Cai and S. J. Turner, *"An Algorithm for Distributed Discrete-Event Simulation - The Carrier Null Message Approach"*, Proc. of the SCS Multiconference on Distributed Simulation, pp. 3-8, January 1990.

[4] A. L. DeCegama, *"The Technology of Parallel Processing - Parallel Processing Architectures and VLSI Hardware"*, volume 1, Prentice-Hall, 1989.

[5] D. M. Dias and J. R. Jump, *"Analysis and Simulation of Buffered Delta Networks"*, IEEE Trans. on Computers, Vol. C-20, No. 4, pp. 273-282, April 1981.

[6] R. M. Fujimoto, *"Parallel Discrete Event Simulation"*, Comm. of ACM, Vol. 33, No. 10, pp. 31-53, October 1990.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderan, *"PVM 3 User's Guide and Reference Manual"*, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1993.

[8] D. R. Jefferson, *"Virtual Time"*, ACM Trans. on Programming Languages and Systems, Vol. 7, No. 3, pp. 404-425, July 1985.

[9] S. C. Kothari, *"Multistage Interconnection Networks for Multiprocessor Systems"*, Advances in Computer, volume 26, pp. 155-199, Academic Press, 1987.

[10] M. Kumar and J. R. Jump, *"Performance of Unbuffered Shuffle-exchange Networks"*, IEEE Trans. Computers, Vol. C-35, pp. 573-578, June 1986.

[11] S. Lakshmivarahan and S. K. Dhall, *"Analysis and Design of Parallel Algorithms, Arithmetic and Matrix Problems"*, McGraw-Hill, 1990.

[12] D. H. Lawrie, *"Access and Alignment of Data in an Array Processor"*, IEEE Trans. on Computers, Vol. C-20, No. 4, pp. 273-282, April 1981.

[13] M. Lee and C. L. Wu, *"Performance Analysis of Circuit Switching Baseline Interconnection Networks"*, Proc. of the 11th Annual International Symposium on Computer Architecture, pp. 82-90, IEEE Computer Society Press, 1984.

[14] J. Misra and K. M. Chandy, *"Asynchronous Distributed Simulation via a Sequence of Parallel Computations"*, Comm. of the ACM, Vol. 24, No. 4, pp. 198–206, April 1981.

[15] J. Misra, *"Distributed Discrete-Event Simulation"*, ACM Computing Surveys, Vol. 18, No. 1, pp. 39-65, March 1986.

[16] J. H. Patel, *"Processor-Memory Interconnection for Multiprocessors"*, Proc. of the 6th Annual Symposium on Computer Architecture, pp. 168-177, April 1979.

[17] A. Pattavina, *"Nonblocking Architectures for ATM Switching"*, IEEE Communications Magazine, pp. 38-48, February 1993.

[18] R. Rooholamini et. al, *"Finding the Right ATM Switch for the Market"*, IEEE Computer, pp.16-28, April 1994.

[19] H. J. Siegal, *" Interconnection Networks for large-Scale Parallel Processing"*, Second Edition, McGraw-Hill, 1992.

[20] S. C. Tay and Y. M. Teo, *"Parallel Discrete-Event Simulation on Distributed-Memory Multicomputers"*, Technical Report TRA3/94, Department of Information Systems and Computer Science, National University of Singapore, March 1994.

[21] C. L. Wu and T. Y. Feng, *"Tutorial : Interconnection Networks for Parallel and Distributed Processing"*, IEEE Computer Society Press, 1984.