Proceedings of the 1995 Winter Simulation Conference
ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman

PARALLEL SIMULATION OF THE IBM SP2 INTERCONNECTION NETWORK

Caroline Benveniste

Department of Electrical Engineering
Columbia University
New York, NY 10027, U.S.A.

Philip Heidelberger

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, U.S.A.

ABSTRACT

Simulations of large multistage interconnection networks used in parallel processing systems are computationally expensive. This paper describes the parallel simulation of a realistic and highly detailed model of the IBM SP2 interconnection network. The challenges involved in efficiently parallelizing such a complex simulation are discussed.

1 INTRODUCTION

Interconnection networks have been identified as suitable candidates for parallel simulation, e.g., see Goli et al. (1989), Konas and Yew (1994), Miguel et al. (1995), Nicol (1988), Yu, Towsley and Heidelberger (1989) and the references therein. Previous papers on this subject have typically modeled the networks at an abstract level, or modeled very simple networks. In this paper, we consider parallel simulation of an accurate and highly detailed model of a real multistage interconnection network used in an actual parallel computer, the IBM SP2. The starting point of this work was an existing sequential simulator of the network developed by one of the authors (Caroline Benveniste). The serial simulator has been used in a number of architectural studies e.g., Benveniste and Hsu (1994), Baylor, Benveniste and Hsu (1994,1995). However, it proved to be quite slow when confronted with simulations of large machines. For example, it could take up to 1 day (on a workstation) to simulate the network interconnecting 128 processors for 1 second of simulation time. This provided motivation for our effort to parallelize the simulator. This effort has resulted in a parallel simulator that runs on the SP2 and produces useful speedups for simulations of a large and complex system.

The rest of the paper is organized as follows. A brief description of the SP2 interconnection network is given in Section 2; a detailed description may be found in Stunkel et al. (1995). Section 2 also describes our general approach in designing the parallel simulator. A detailed description of our synchronization algorithm and several optimizations are given in Section 3. We report on the performance of the parallel simulator in Section 4 and give conclusions in Section 5.

2 MODEL AND GENERAL APPROACH

The SP2 interconnection network consists of interconnected racks. Each rack is a 16×16 bidirectional switch board. A switch board contains 8 switching elements arranged in 2 stages with 4 elements in each stage. The structure of a switching element is shown in Figure 1.

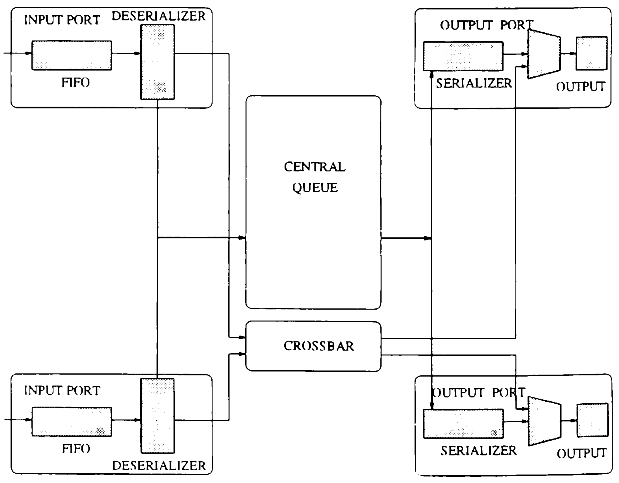

Figure 1: The SP2 High Performance Switch

A message on the SP system is broken up into self-routing packets. Packets can be up to 255 flow-control digits (or flits, 1 flit = 1 byte in this case) in length. When a packet enters a switch the first flit of the packet is stored in the input buffer. In the next cycle the first flit will move to the deserializing buffer. When the second flit, which contains the routing information, has entered the deserializing buffer, the switch determines to which output the packet will be transferred. The logic will then decide whether the packet can use the crossbar and thus take a fast path through the switch, or whether it must be stored in the central queue. The packet will be stored in the central queue if another packet is already using the desired output.

The data width in and out of the central queue is 8 flits, or one chunk. The switch was designed in this way so that the bandwidth in and out of the central queue would equal the bandwidth elsewhere in the switch. Since there are 8 inputs and 8 outputs in each switch, each input or output will be able to transfer one chunk in/out of the central queue every 8 cycles. The arbitration at the input and output of the central queue is LRU.

The flow of each flit through the network is simulated. The simulator keeps track of where flits are located and moves, or tries to move them on each cycle. Time periods without any network activity are skipped over; the simulator advances the global time to the first time at which network activity resumes.

Our simulator is trace-driven, i.e., the simulator reads message arrival times, message lengths, and source-destination pairs from an input file. This input file can be recorded by observing, or tracing, an actual SP2 system, although the results in this paper report on synthetically generated traces. Because the serial simulator was so detailed, we decided not to start from scratch but rather to add parallelism to the existing simulator. This affected a number of design decisions, including the choice of synchronization algorithm and the way in which the model was partitioned onto the parallel processor.

For example, adding the code to support rollbacks as required by an optimistic parallel simulation implementation would have been extremely difficult (and probably computationally expensive), given the existing data structures and algorithms. Thus we chose a conservative synchronization algorithm. Because of the high interconnectivity of the SP2 network, protocols based on pair-wise appointments or null messages seemed to offer little advantage over simpler window-based protocols. Thus we chose the YAWNS windowing protocol (see Nicol, Micheal and Inouye 1989 and Nicol 1993) as the basis of our synchronization algorithm. In a windowing algorithm, the time of the next window represents the earliest time at which some processor's state may be affected by an event on another processor. The window length is constructed so that messages sent during one window do not affect the state of another processor until the next window. At the end of the window, all messages are received and the next window length is computed.

Since the building block of the SP-machines is the rack, a natural partition is to assign one rack to one physical processor. This fits in nicely with the existing simulator's data structures. It results in a reasonable computation to communication ratio and provides for a useful amount of potential parallelism; a 64 node system has 4 racks while a 128 node system has 12 racks. While different partitioning policies are possible, in this paper we assume that the simulator has been configured in this way. With this partitioning, packets that are generated on one rack, and have their destination on the same rack can be completely simulated by one processor. However, packets crossing rack boundaries require interprocessor communication. Because routes are statically determined, it is possible to tell exactly which racks a packet flows through at the time the packet is read from the trace file.

Given this type of partitioning, processors need to communicate when flits leave, or try to leave the output buffer on one rack to enter an input buffer on another rack. If the input buffer is full, then the flit is blocked from leaving the output buffer. It would have been costly to implement a handshaking protocol on each flit transfer to request a slot in the input buffer. In addition, since input buffers are reasonably large (31 flits), most flits are successfully transferred. We avoided the necessity for such a protocol by making shadow copies of the output buffers (on different racks) that feed the input buffers of a processor's rack. The states of the shadow buffers are kept consistent by messages that are received at window boundaries. To avoid the handshaking protocol, the window is constructed so that all flit transfer requests within the window will be satisfied (provided no buffers are already full). The window calculation thus includes an estimate of the earliest time at which some buffer may become full.

Once a chunk header enters an output buffer, all flits in that chunk will be transferred in consecutive cycles (provided the input buffer is not full). Thus our initial algorithm kept track of chunk headers and computed a window when each chunk header might leave the rack. Messages reflected information about when the first flit of a chunk exits the rack. However, this led to small windows, and excessive message passing. Furthermore, once the header of a *packet* leaves an input buffer, the subsequent flits of the packet typically follow at regularly spaced intervals. Thus only the packet headers need be accounted for in the window algorithm, plus exceptions to the assumption that flits are regularly spaced. These exceptions only occur in certain circumstances associated with full buffers. Further details and optimizations of the window algorithm are given in Section 3.

Our simulator was implemented using the MPI (message passing interface) standard communications library and runs on the SP2 parallel processor.

3 SYNCHRONIZATION DETAILS

We first describe our basic synchronization algorithm, and then an optimized algorithm that led to improved simulator performance.

3.1 Basic Algorithm

The first algorithm we implemented was based on the location of chunk headers. The window calculation took the minimum of three numbers to predict the first time at which a communication between racks will be necessary. The first number is the minimum time at which the head of a chunk may leave a rack. To calculate this quantity, the simulator keeps track of where the chunk headers are in the switches. The switch has a number of internal registers: an input FIFO, a deserializing buffer, a central queue, a serializing buffer, and an output buffer. In addition, the racks contain two stages of switching elements. Therefore, depending on which internal register and which stage the chunk header is in, there is an associated minimum time at which the chunk header may cross a rack boundary. Rather than check each buffer at the window calculation to see which chunks may cross a rack boundary first, an array is kept of how many chunk headers are at each location. In the window calculation, the location with the smallest delay and a non-zero number of chunks associated with it will contribute to the window calculation.

The second number used in the window calculation is the time at which the next packet that has not yet entered the network can leave the rack. This is necessary because initially there are no packets in the network so this number will be used to calculate the first window. Also, during the run the new packets that are being injected into the network may eventually leave the source rack and, if there are no other packets in the network that cross rack boundaries, this number may be the minimum window time.

The final component in the window calculation is the full buffer time. This is the minimum time at which some input buffer in a switch may become full. This quantity enables us to avoid communication on each flit transfer. To compute this quantity, we note the number of flits in each shadow buffer (say x_i for buffer i) and if this quantity is positive then we compare that with the number of free slots in the corresponding input buffer (say y_i). If $x_i < y_i$, then there is no possibility that buffer i will fill during the next window. However, if $x_i \geq y_i$, then buffer i could fill in y_i cycles. The full buffer time is then simply the minimum of all y_i's such that $x_i \geq y_i$. If one of the buffers is full at a window calculation, information is exchanged identifying which buffers are full, thereby keeping the states of the shadow buffers on a receiving rack consistent with the states of the output buffers on a sending rack. During such a full buffer exchange, for each pair of communicating processors i and j, processor i sends (in one message) the identities of all full input buffers fed by processor j. The window length is then set to be one cycle, since the time that the input buffer will become unblocked is not known. In addition, during that window, no flits are transfered to a full input buffer. This basic algorithm achieved a speedup of 1.9 on 4 processors when simulating a 64 node system with 200 flit messages and an expected utilization of 0.2 per port.

3.2 Optimized Algorithm

To improve the performance of the basic algorithm, we implemented a number of optimizations. In the basic algorithm, each window calculation involved two global reductions. The first reduction counts the number of messages sent to each processor. Processors then poll for messages until they have received all messages sent to them. These messages are processed, and then a second reduction calculates the next window time. The first optimization used only one reduction per window, computing both the message counts and the next window time in a single reduction. With this implementation, processors have less information available to them at the time the reduction is started (since they haven't necessarily picked up all their messages yet). For example, a processor may receive a message about a particular chunk after the window calculation. The processor may eventually have to send this chunk to still another processor. The possibility of that chunk causing the next window must be accounted for. We did so by placing the burden on the sending processor; if, during a window, processor i sent a chunk to processor j that eventually must be sent to processor k, processor i must ensure that the next window length is no larger than the minimum time for the chunk to flow through processor j. In addition, the full buffer time calculation had to be modified since a newly arriving, but unprocessed, chunk to an empty shadow buffer could cause a full buffer if there are fewer than 8 slots available in the destination input buffer. (The basic algorithm only computes full buffer times for non-empty shadow buffers.) Similarly, a newly arriving, but unprocessed, chunk to an input buffer with only one available slot causes that buffer to be full at the start of the next cycle. With this optimization, the speedup increased from 1.9 to 2.3.

We next optimized the code to have communication occur between racks only at the beginning of packets, rather than at the beginning of chunks. This was possible since, typically, once a packet starts transmitting it transmits 1 flit each cycle without interruption as long as the receiving input buffer does not fill. Therefore, during normal operation, each chunk does not need to be sent, and only the heads of packets are explicitly sent.

However, one situation may arise that invalidates the assumption that once a packet begins transmission out of a rack, it will be transmitted without interruption. This can occur because of the least-recently-used (LRU) algorithm used for arbitration out of the central queue. Usually, a packet will transmit 8 flits every 8 cycles. However if one packet is transmitting out of an output every 8 cycles, and one of the other outputs of that switch becomes blocked because an input buffer is full, the blocked output will move to the front of the LRU queue. If it should happen that the time at which it becomes unblocked corresponds to the time that the first output was ready to transmit, the second (formerly blocked) output will get priority and the first output will transmit one cycle late. This information must be communicated to the receiving node, otherwise there will be an inconsistency in the system. In the worst case all 7 outputs will be blocked and unblocked within 7 cycles starting at the cycle in which the only unblocked output was scheduled to transmit. Since the other 7 outputs were blocked they will have priority over non-blocked output, and the non-blocked output may be delayed for 7 cycles. To handle this situation we identify this period, which, in the worst case will be 7 cycles. During this period we cause a window to occur every cycle. In addition, during this period, we notify the receiving processor each time a chunk is sent, rather than just when the head of the packet is sent. By doing this we guarantee that the parallel nodes will have consistent states. This subtle situation was not identified and corrected until we noticed a slight discrepancy between the results of the serial and parallel simulators.

As a final optimization, we took certain queuing delays into account to further increase the window length. Specifically, we added a fourth number to the reduction in order to prevent windows from occurring when a packet is blocked in the central queue. Consider the following situation: a packet is in the process of transmitting out of the central queue for a particular output, and another packet is blocked behind it. Thus, the second packet would cause a window to occur every x cycles where x is the time for the head of the packet blocked in the central queue to leave the rack. However, since the output is being used we do not want to cause a window in x cycles, but rather in x + the minimum time for the previous message to finish transmitting out of the central queue. We have implemented this optimization in the following manner: each time a packet is added to the central queue, we check to see if another packet is blocking it. (This packet may be using the same output of the central queue, or it may be using the crossbar.) If there is, we then take the minimum time remaining to transmit this packet, add to it the delay for the current packet to leave the rack after it becomes unblocked, and place this number on a heap. At each window calculation, the number at the top of the heap is used in the window calculation since this is also a time when the head of a packet may cross rack boundaries. A packet is removed from the heap when it becomes unblocked and begins transmitting out of the central queue. It is possible that the previous packet has become delayed, and does not complete transmission at the predicted time. Therefore, at each window calculation any packets on the heap whose time is greater than or equal to the current simulation time will be removed from the heap and re-inserted on the heap with a new estimated time.

With these optimizations, the speedup increased from 2.3 to 2.6.

4 SIMULATOR PERFORMANCE

We used synthetically generated traces to drive the parallel and serial simulators. The communication pattern had a uniform random distribution. The (time) separation between messages at each node was exponentially distributed with a mean that varied with the applied load and message length. We ran simulations of 64 and 128 processor SP2 systems, and compared the results of the parallel simulator to that of the serial simulator for both system sizes. All parallel simulations of the 64-processor system were run on 4 SP2 nodes, and all simulations of the 128-way processor system were run on 12 SP2 nodes. The input traces have messages generated during 1 million cycles.

Message Size (flits)	200	1K	2K	8K
Speedup	2.62	2.06	1.70	1.75
Average Window Size	7.56	2.02	1.12	1.01
FBE Window Fraction	0	0.75	0.95	0.996

Table 1: Results for Simulations of a 64-Processor SP2 with an Applied Load of 0.2

In Table 1 we present the results for the 64 way system with an applied load, i.e., average port utilization, of 0.2 but with different message sizes. The first row shows the speedup of the parallel simulator over the serial simulator for the different messages sizes. The second row shows the average window size (in cycles) in the parallel simulator, and the third row shows the fraction of windows that were caused by a full buffer exchange. The greatest speedup was obtained for the simulation run with 200 flit messages. In this simulation the window size was over 7 cycles, and no full buffer exchanges took place. With the larger message sizes the network was more stressed because the traffic is burstier, and this causes the switch buffers to fill. In the three runs with larger message sizes the windows are smaller and the fraction of windows caused by a full buffer exchange also increases so that for 8K messages almost all windows are caused by a full buffer exchange. However, the speedup of the simulator when run with 8K messages is slightly better than the speedup with 2K messages. This can be understood by considering that with the 8K messages more activity is occurring on the network at certain times, and this activity is now divided

Message Size (flits)	200	1K	2K	8K
Speedup	5.34	4.04	3.56	3.78
Average Window Size	3.825	1.27	1.002	1.01
FBE Window Fraction	0	0.86	0.999	0.998

Table 2: Results for Simulations of a 128-Processor SP2 with an Applied Load of 0.2

among 4 processors on the parallel simulator. There is not much difference in window size between the 2K run and the 8K run, but in the 8K run there is more activity (more flits moving) within each window. The speedup of the run with 1K messages is slightly better than the 2K and 8K runs because the window size is larger and a smaller percentage of the windows are caused by full buffer exchanges.

In Table 2 we present results for the simulation of a 128-way SP2 with different message sizes and an applied load of 0.2. These results are similar to the ones presented for the 64-way SP2. However, the average window size is smaller for 200 flit messages than it was for the corresponding simulation of the 64-processor system. This is caused by the fact that when there are no full buffers in the system, the windows are determined by the next time that the head of a packet leaves a rack. Since there are more racks and more packets in the 128-way system than in the 64-way system, there will be more instances when the head of a packet is about to leave a rack. Still, the speedup of the simulator with 200 flit messages is almost 5.5.

In Table 3 we present the results of a simulation run of the 64-processor system with 200 flit messages, and an applied load varying from 0.2 to 0.5. The best

Applied Load	0.2	0.3	0.5
Speedup	2.62	2.83	2.75
Average Window Size	7.56	5.39	2.03
FBE Window Fraction	0	0.02	0.54

Table 3: Results for Simulations of a 64-Processor SP2 with 200 Flit Messages

speedup is obtained for an applied load of 0.3. With that applied load the network is moderately loaded, but still able to transfer packets with few full buffers occurring. The speedup is better with a load of 0.3 than with a load of 0.2 because there is more network activity which can be successfully parallelized. The window sizes for the two runs are similar, but more work is being done during each window in the experiment with the higher load. When the load is increased to 0.5, the number of full buffers increases rapidly, which in turn causes a much smaller average window size, and the speedup declines.

In Table 4 we present results for simulations of a 128-way system with 200 flit messages and varying applied loads. The results are similar to those presented for the 64-way simulations. In this case the best speedup is obtained with an input load of 0.3. For this simulation, a speedup of over 6 was obtained.

Applied Load	0.2	0.3	0.5
Speedup	5.34	6.17	5.71
Average Window Size	3.83	2.79	1.53
FBE Window Fraction	0	0	0.38

Table 4: Results for Simulations of a 128-Processor SP2 with 200 Flit Messages

	Speedup	Average Window Size	FBE Window Fraction
0%	2.62	7.56	0
2%	2.62	7.56	0
4%	2.65	7.49	0.01
6%	2.59	6.98	0.08
7%	2.47	4.86	0.37
8%	1.87	1.006	0.999

Table 5: Results for Simulations of a 64-Processor SP2 with a Skewed Load

In Table 5 we show the results of simulations of a 64-processor SP2 system with a skewed input load. The skews are given as the percent of the total load that is going to simulated processor 0. This creates a workload imbalance for the parallel simulator and, for a given overall input load, causes a greater fraction of full buffers due to the "hot-spot" effect (see Pfister and Norton 1985). These simulations were run with 200 flit messages, and a total input load of 0.2. With skews of 6% and less there is little effect on the speedup of the parallel simulator, and the average window size with a skew of 6% is still relatively high at 6.98 cycles. However, starting with the skew of 7% the window size drops, and the percentage of windows caused by a full buffer exchange rises steeply. With a skew of 8% the window size has dropped to just over 1 cycle, and almost all the windows are caused by a full buffer exchange. The speedup of this last simulation has dropped to 1.87.

5 CONCLUSIONS

This paper describes a parallelization of a realistic and highly detailed model of the SP2 interconnection network. The model runs on the SP2, producing useful speedups of up to just over 6 on 12 processors. Speedups are affected by the overall level of network activity as well as the frequency of full buffer blocking events that span processor boundaries.

Given that we started with an existing sequential simulator, we had anticipated that it would be rather straightforward to add the necessary synchronization algorithms and message passing code. However, our initial implementation was slow due to excessive synchronization and message passing. Improving performance required rather more sophisticated techniques, as well as a deeper understanding and exploitation of the structure of the network. In addition, certain situations caused slight divergences between the results of the serial and parallel simulators that were difficult to track down (the flits not coming out at regular intervals under some blocking conditions). One of the advantages of simulation is the flexibility to model a variety of complex situations. While we have succeeded in developing a parallel simulator of a particular, highly complex "real world" system, the parallelization relies heavily on the structure of the network. Thus adapting the parallel simulator to model alternative network designs is not necessarily straightforward. We are, however, interested in using the parallel simulator in production runs for specific modeling studies.

ACKNOWLEDGMENTS

We are grateful to David Nicol for letting us use his heap manipulation routines. We would also like to thank Craig Stunkel for answering all our network questions. The work of Caroline Benvensite was funded under a work/study program with the IBM T.J. Watson Research Center.

REFERENCES

Baylor, S., C. Benveniste, and Y. Hsu. 1994. Performance evaluation of a massively parallel I/O subsystem. In *Proceedings of the 8th International Parallel Processing Symposium, Workshop on Input/Output in Parallel Computer Systems*, 1-15, IEEE TCPP and ACM SIGARCH.

Baylor, S., C. Benveniste, and Y. Hsu. 1995. performance evaluation of a parallel I/O architecture. In *Proceedings of the International Conference on Supercomputing*, 404-413, ACM SIGARCH.

Benveniste, C. and Y. Hsu. 1994. Performance evaluation of central queue arbitration policies for the Vulcan parallel system. In *Proceedings of the 1994 Summer Computer Simulation Conference*, The Society for Computer Simulation.

Goli, P., P. Heidelberger, D. Towsley, and Q. Yu. 1990. Processor assignment and synchronization in parallel simulation of multistage interconnection networks. In *Distributed Simulation*, 181-187, The Society for Computer Simulation International.

Konas, P. and P-C Yew. 1994. Improved parallel architectural simulations on shared-memory multiprocessors. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS)*, 32-38, IEEE Computer Society Press.

Miguel, J., A. Arruabarrena and R. Beivide. 1995. Conservative parallel simulation of a message-passing network: a performance study. In *Proceedings of the 1995 Summer Computer Simulation Conference*, 825-830, The Society for Computer Simulation.

Nicol, D.M. 1988. Parallel discrete-event simulation of FCFS stochastic queuing networks. In *Proceedings ACM/SIGPLAN PPEALS 1988: Experiences with Applications, Languages and Systems*, 124-137, ACM Press.

Nicol, D.M. 1993. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, 40: 304-333.

Nicol, D.M., C. Micheal, and P. Inouye. 1989. Efficient aggregation of multiple LP's in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference*, 680-685, IEEE Computer Society Press.

Pfister, G.F. and A.N. Norton. 1985. "Hot-Spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers* 34: 943-948.

Stunkel, C.B., D.G. Shea, B. Abali, M.G. Atkins, C.A. Bender, D.G. Grice, P. Hochschild, D.J. Joseph, B.J. Nathanson, R.A. Swetz, R.F. Stucke, M. Tsao, and P.R. Varker. 1995. The SP2 high-performance switch. *IBM Systems Journal* 34: 185-204.

Yu, Q., D. Towsley, and P. Heidelberger. 1989. Time-driven parallel simulation of multistage interconnection networks. In *Distributed Simulation, 1989*, 191-196, The Society for Computer Simulation International.

AUTHOR BIOGRAPHIES

CAROLINE BENVENISTE received an A.B. in physics from Harvard University in 1983. She is currently a Ph.D. candidate in Electrical Engineering at Columbia University. She received an IBM graduate fellowship award and has also worked part time at the IBM T.J. Watson Research Center under a work/study program while pursuing her degree.

PHILIP HEIDELBERGER received a B.A. in mathematics from Oberlin College in 1974 and a Ph.D. in Operations Research from Stanford University in 1978. He has been a Research Staff Member at the IBM T.J. Watson Research Center since 1978. While on sabbatical in 1994-1995, he was a visiting scientist at Cambridge University and at ICASE, NASA Langley Research Center. He is an Area Editor of *ACM TOMACS*, was program chairman of the 1989 Winter Simulation Conference, and program co-chairman of the ACM Sigmetrics/Performance '92 Conference. He is a Fellow of the IEEE.