

# On Measuring the Performance of Adaptive Wormhole Routing

Loren Schwiebert  
Dept. of Electrical & Computer Engg.  
Wayne State University  
Detroit, MI 48202–3902  
loren@ece.eng.wayne.edu

D. N. Jayasimha\*  
Intel Corporation, MS RN2-02  
2200 Mission College Blvd.  
Santa Clara, CA 95052  
djayasim@mipos2.intel.com

## Abstract

*Adaptive routing is widely regarded as a promising approach to improving interconnection network performance. Many designers of adaptive routing algorithms have used synthetic communication patterns, such as uniform and transpose traffic, to compare the performance of various adaptive routing algorithms with each other and with oblivious routing. These comparisons have shown that the average message latency is usually lower with adaptive routing. On the other hand, when a parallel program is executed on a multiprocessor, the goal is to reduce the total execution time. In this paper, we explain why improving the average message latency of a routing algorithm does not necessarily lead to a lower execution time for real applications. We support this observation by reporting simulation results for both adaptive and oblivious routing using communication derived from real applications. Specifically, we report the performance of various routing algorithms for directed acyclic graphs (DAGs) derived from the Cholesky factorization of sparse matrices. Our results show that there is little correlation between average message latency and the total execution time of a parallel program. Hence, average message latency does not seem to be a useful measure of the performance of a routing algorithm. This strongly suggests that current comparisons of routing algorithms do not provide a reliable indication of the performance improvements to be realized by executing programs on a multiprocessor with such a routing algorithm. We interpret these results and suggest several alternatives for further research.*

## 1 Introduction

Wormhole routing [4] has become the switching technique of choice in distributed-memory multiprocessors. Implementations of wormhole routing divide each message into flits. The header flit of a message contains the routing information and the data flits of the message follow the

header flit through the network. When the header arrives at an intermediate router, the router immediately forwards the message header to a neighboring router if an output channel the message can use is available.

Since the flits of a message are forwarded as soon as possible, the message latency is largely insensitive to the distance between the source and destination. In addition, wormhole routing requires only enough storage on a router to buffer a few flits, rather than the entire message. These two properties account for the popularity of wormhole routing in distributed-memory multiprocessors.

Although wormhole routing can reduce the communication overhead in large-scale multiprocessors, wormhole routing does not provide a complete solution to the problem of minimizing communication overhead in parallel programs [2]. The primary drawback to wormhole routing is the contention that can occur with even moderate traffic, which leads to higher message latency and increased communication overhead. Channel contention can be reduced through software techniques such as *mapping*, hardware techniques such as *adaptive routing*, or a combination of both techniques.

### 1.1 Hardware Techniques

A cost-effective method of reducing message latency, proposed by Dally [3], is to allow multiple *virtual* channels to share the same physical channel. Each virtual channel has a separate buffer, with multiple messages multiplexed over the same physical channel. Both latency and contention can be further reduced by using the multiple paths between the source and destination nodes. Many adaptive routing algorithms have been proposed to exploit this possibility.

The simplest routing algorithms are *oblivious* and define a single path between the source and destination. Adaptive routing algorithms, on the other hand, support multiple paths between the source and destination. Minimal routing algorithms allow only shortest paths to be chosen. Adaptive routing algorithms can be further differentiated by the fraction of shortest paths they allow. *Fully adaptive* routing algorithms allow all messages to use any shortest

---

\*This work was done while this author was with the Department of Computer and Information Science at Ohio State University in Columbus, Ohio.

path. Some fully adaptive routing algorithms allow more adaptiveness than others by placing fewer restrictions on the choice of channels.

Many researchers have proposed adaptive routing algorithms to address the problem of contention. The hypothesis is that adaptive routing improves network throughput and performance by allowing messages to route around congested channels. This claim has been substantiated by comparing the average message latency among routing algorithms. With few exceptions, the trend has been that increasing adaptiveness results in lower average message latency, even when the additional complexity of adaptive routing [1] is included in the comparison [9]. These comparisons are based on traffic patterns such as transpose and uniform traffic, which may not adequately reflect the message traffic seen in typical parallel programs.

## 1.2 Software Techniques

*Average* message latency does not correlate well with the execution time of a parallel program. The reason there is not much of a relationship between average message latency and execution time becomes clear once one considers how the communication among processors affects the execution time.

Before executing a parallel program, the program is first decomposed into tasks. The tasks each require some computation time. Tasks may also require messages from other tasks prior to execution and may transmit results to other tasks after execution. The mapping problem addresses the problem of assigning these tasks to the processors so that the parallel program executes as quickly as possible.

Early work on the mapping problem focused on reducing the total communication cost. (Please see [12] for references on mapping.) Reducing the communication cost, however, does not necessarily reduce the total execution time, because *the impact on total execution time depends on which messages are delayed*. This is due to what is commonly referred to as the critical path. The *critical path* is the longest path, counting both computation and communication costs, from a source node of the task graph to the last node of the task graph. Delaying a message on the critical path, a critical edge, increases the total execution time of the task graph. Other messages can become critical edges if they are delayed too long.

The average message latency has been used to measure the performance of a routing algorithm, but the performance of a parallel program is determined by the total execution time. The usual unstated assumption is that a lower average message latency results in a lower total execution time, but that is not necessarily the case because of critical edges. The critical edges are typically a small subset of the edges in a parallel program. Thus, the average message latency has little relationship to the communi-

cation cost of critical edges. In the remainder of this paper, empirical evidence is presented to support this argument.

## 2 Program Characteristics

In order to determine whether average message latency provides a good indication of execution time or the critical edges play a major role in determining execution time, we simulated some parallel programs. Each parallel program used in the simulation is represented using the directed acyclic graph (DAG) model. The DAGs used in this paper are derived from the Cholesky factorization of three different irregular sparse matrices taken from real applications [5]. To provide differing computation-to-communication ratios, CPU clock cycle times of 20 nanosecs and 5 nanosecs are used. DAGs 1, 3, and 5 execute on the slower processors; the corresponding DAGs on the faster processors are 2, 4, and 6. The characteristics of each DAG are shown in table 1. Each message is prepended by an additional one-byte header during simulation. This header contains the destination address, which is used by all the routing algorithms to determine the path from the source to the destination. To reduce the simulation time while maintaining the original DAG structure, the task weights and message lengths have been proportionally reduced from those in the original DAG [5]. The message start-up times is also correspondingly reduced to ten clock CPU cycles. We assume a communication coprocessor is available to transmit messages with no further interaction with the CPU. If a task sends more than one message, these messages are generated at intervals of ten clock cycles to simulate the start-up time of each message. The communication characteristics of a DAG change when the tasks are assigned to the processors, because the source and destination tasks are sometimes placed on the same processor. The number of messages and their length characteristics are also shown in table 1. The execution time of the tasks affects the task assignment, so a different cycle time changes the number of injected messages.

Table 1: DAG Characteristics

Input			Injected Msgs	
DAG #	# of Tasks	Task Weight Min/Avg/Max	# of Msgs	Msg Length Min/Avg/Max
1	469	9/13.0/21	429	14/18.9/26
2	469	9/13.0/21	468	14/19.0/26
3	659	9/22.2/45	795	14/28.8/50
4	659	9/22.2/45	811	14/29.0/50
5	3839	9/14.4/33	3783	14/21.2/38
6	3839	9/14.4/33	3904	14/21.3/38

### 3 Simulation Methodology

The mapping is performed by a simulator, referred to as *the mapper*, which accurately computes the execution time of the mapping. The mapper has an interface to a network simulator [8], referred to as *the simulator*, which was developed using the CSIM [11] simulation package. The network simulator permits a flit-level simulation of the interconnection network. The experiments in this paper use demand multiplexing. Both hypercube and 2D meshes are simulated. Due to limited space, only the results for the 2D mesh are presented here. The hypercube results, which are quite similar, are presented in [14].

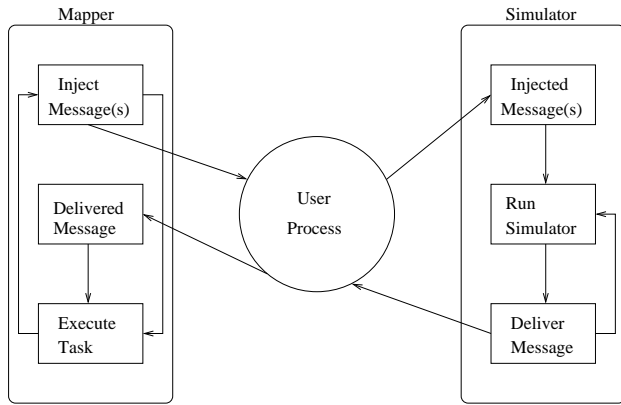


Figure 1: Interaction between the Mapper and Simulator

The simulation is event-driven, with each message corresponding to a unique event. The simulator allows different message traffic patterns to be specified by modifying a single process, called the *user process*. For these simulations, the user process must support message traffic that is generated by a parallel program. This requires interaction between the mapper and the simulator as depicted in figure 1, which shows the high-level operation of the user process. In essence, the user process provides the connection between the processors and the network. When a task completes execution, the messages sent by this task are injected into the network. The user process passes these injected messages to the simulator. The simulator then routes these messages. When a message is delivered, the delivery time is passed to the user process, which in turn passes this information back to the mapper. The mapper uses this information to update the status of the destination task. If all the messages have arrived for this task and the processor is free, then the mapper initiates this task and subsequently injects additional messages into the network.

#### 3.1 Mapping Heuristics

Two different mapping heuristics are used. The heuristic proposed by Yang, Bic, and Nicolau [15] and referred to

as *the YBN heuristic*, determines the mapping independent of contention in the interconnection network. The mapping is generated independent of the routing algorithm, so a comparison of different routing algorithms with the same mapping is possible.

The other heuristic, which was proposed by Schwiebert and Jayasimha [12] and is referred to as *our heuristic*, produces a mapping by iteratively adjusting the previous mapping. Each mapping is simulated in order to determine the network contention., which is used to generate an alternative mapping. Different mappings are produced for each routing algorithm, which optimizes the mapping for that routing algorithm.

Both mapping heuristics require the input DAG to be a clustered task graph with one cluster per processor. The Dominant Sequence Clustering (DSC) heuristic proposed by Yang and Gerasoulis [17] is used to cluster the task graphs and the cluster merging algorithm [6] implemented in the Pyrrhos [16] system is used to produce the correct number of clusters. After the clustering and merging steps, there is one cluster per processor. We calculate the latest starting time of each task and execute the ready task with the minimum latest starting time first. A task becomes ready once it has received all of its messages. The messages generated by a task are prioritized by injecting them into the network in increasing order of the latest starting time of the destination tasks.

The interaction between the simulator and the mapper allows the contention experienced by messages to be relayed to our mapping heuristic. Based on the feedback from each simulation, the mapping is modified by moving clusters with critical edges to processors that are topologically closer. The number of different mappings that are tried is at most the number of processors. Since different routing algorithms experience different contention patterns, our mapping heuristic adjusts automatically to match the particular characteristics of each routing algorithm. The YBN heuristic does not use contention information to select a mapping.

#### 3.2 Routing Algorithms

Two fully adaptive routing algorithms and an oblivious routing algorithm have been used in these experiments. Only minimal routing is used. The fully adaptive routing algorithms are opt-y [13] and mad-y [7]. Opt-y is more adaptive than mad-y [13]. Adaptive routers are more complex than oblivious routers [1]. The calculation of network cycle time based on router complexity permits a more realistic comparison of the performance of different routing algorithms. A network cycle time of 5.92 nanosecs is used for the adaptive routing algorithms and 5.41 nanosecs is used for oblivious routing.

## 4 Simulation Results

The total execution time (in nanoseconds) of each DAG, averaged over three runs, is presented. In addition to measuring the execution time of the DAG, the average message latency is also computed.

### 4.1 2D Mesh Simulations

Simulations were run for both  $4 \times 4$  and  $8 \times 8$  meshes with similar results. Due to space constraints, we present results for only the  $8 \times 8$  mesh. The simulation results are shown in table 2. Because of the lower clock cycle time, XY (oblivious) routing is usually best.

Table 2:  $8 \times 8$  Mesh Results (in ns)

DAG	Our Mapping Heuristic			Yang, Bic, and Nicolau		
	Opt-y	Mad-y	XY	Opt-y	Mad-y	XY
1	16404	16352	15589	17557	17267	16896
2	39942	39749	39199	41086	41086	40356
3	21329	21095	20469	22674	22311	22826
4	53159	52786	51764	53865	53485	52643
5	51561	52778	46177	57693	61561	49212
6	84896	85379	83951	87419	87846	86165

Our mapping heuristic always produces a better mapping than the YBN mapping heuristic, because our heuristic adjusts the mapping to reduce contention. Our mapping heuristic always chooses different mappings for each of the routing algorithms, so we are able to address the contention characteristics of each routing algorithm individually.

A scatter plot of the execution times and average message latencies of 64 different mappings of DAG 6 are shown in figures 2 – 4. Similar results were obtained for the other DAGs. The figures show little relationship between the average message latency and the execution time. The correlation coefficients<sup>1</sup> for opt-y, mad-y, and XY are 0.12, 0.51, and 0.14, respectively, suggesting that average message latency is a poor predictor of total execution time.

Table 3 shows the execution times using the same cycle time for all the routing algorithms. The adaptive routing algorithms now often perform better than oblivious routing. When oblivious routing is better, the difference is much less than the difference shown in table 2.

The average message latency shows a lack of correlation with the execution times. For example, with DAG 4 using the YBN heuristic (the same mapping), the average message latency is 247ns with opt-y, 242ns with mad-y and 251ns with XY. In other words, XY performs slightly better even though the average message latency is higher.

<sup>1</sup>A correlation of 1.0 indicates that two things are completely correlated and a correlation of 0.0 would indicate no correlation.

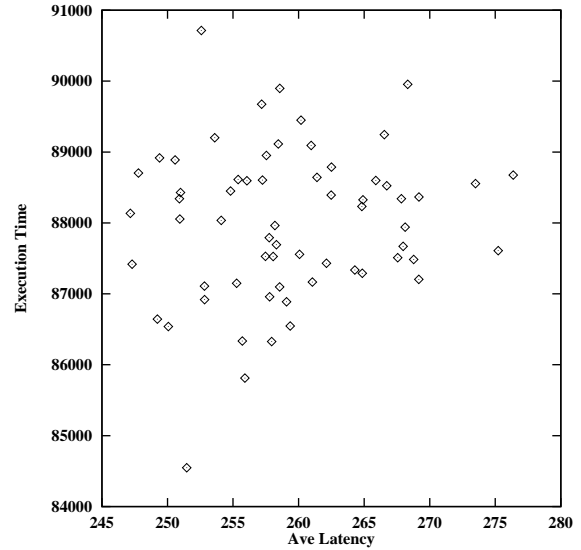


Figure 2: Average Latency vs Execution Time for the Opt-y Routing Algorithm

Table 3:  $8 \times 8$  Mesh with Identical Cycle Times (in ns)

DAG	Our Mapping Heuristic			Yang, Bic, and Nicolau		
	Opt-y	Mad-y	XY	Opt-y	Mad-y	XY
1	15620	15503	15463	16493	16396	16749
2	39932	39094	39342	40356	40356	40356
3	20056	20097	20590	20825	21014	22646
4	52231	51853	52090	53343	52781	52722
5	47772	49230	46696	53306	56237	50231
6	83130	83826	84109	85382	86162	85859

A similar result can be seen with our mapping heuristic. For example, with DAG 5 the average message latencies with opt-y, mad-y, and XY are 939ns, 1154ns, and 973ns, respectively, even though the performance is worse with opt-y than XY. On the other hand, sometimes the average message latency is lower along with the execution time. For example, with DAG 6 using our heuristic, the average message latencies with opt-y, mad-y, and XY are 221ns, 233ns, and 240ns, respectively.

## 5 Discussion and Conclusions

We have explained why the average message latency is a poor predictor of performance in a real parallel processor. The simulation results support our arguments. If the multiprocessor is used in a multiprogramming environment, then the total execution time of individual applications may not be as important as increasing the through-

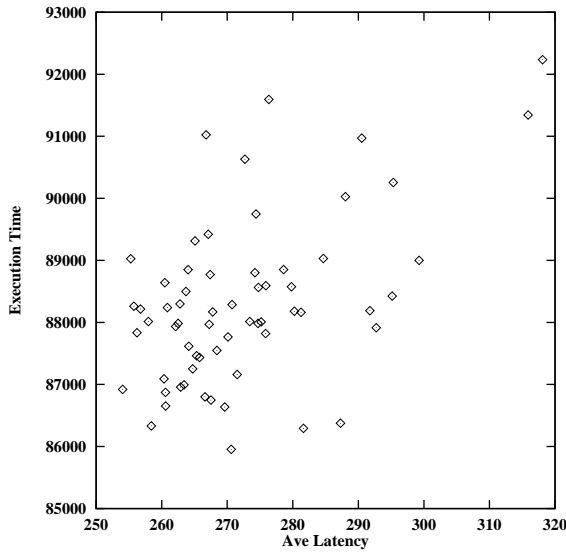


Figure 3: Average Latency vs Execution Time for the Mad-y Routing Algorithm

put of the interconnection network. Most current multiprocessors, however, are either used in single-program mode or are space-shared among applications ensuring that each program has a subset of the processors.

Although only a small number of DAGs were used, there is good reason to believe that the results will hold with a larger number of DAGs. The reason for the relatively poor performance of adaptive routing was not because the chosen DAGs had communication patterns that were better suited for oblivious routing. In fact, the results show exactly the opposite; adaptive routing usually had *lower* average message latency. The delay of critical edges was the cause of poor performance and most parallel programs have a relatively small set of critical edges.

Furthermore, DAGs with a different computation-to-communication ratio are unlikely to produce substantially different results. DAGs with little communication experience less contention, so the routing algorithm makes even less of a difference. The delay of a few critical edges, however, could still result in a mismatch between average message latency and total execution time. DAGs with more communication experience more contention, so the difference in average message latency between adaptive and oblivious routing would increase. The total execution time, however, would still be relatively independent of the average message latency, because the critical edges are a small subset of the messages in most DAGs.

Several topics for future work are currently being explored. One possibility for improving performance is assigning higher priority to critical edges, with lower prior-

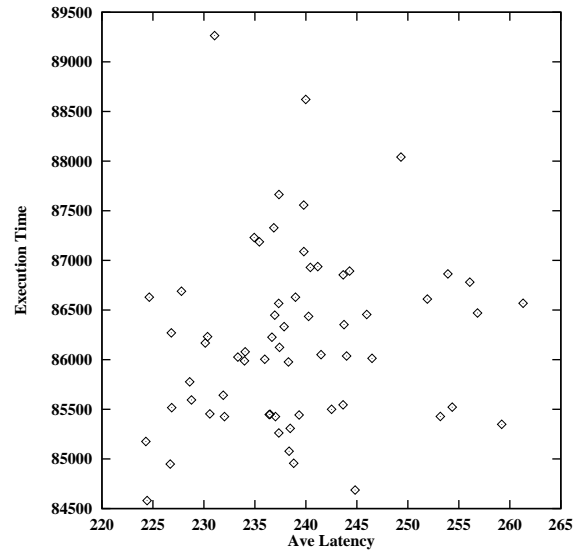


Figure 4: Average Latency vs Execution Time for the XY Routing Algorithm

ities assigned to the other messages depending on when each needs to arrive at the destination. Routing algorithms designed to use these priorities could improve the total execution time, perhaps at the expense of the average message latency. Even at run-time, it should be possible to partially determine priorities.

The interaction between adaptive routing and mapping affects the performance of the routing algorithms. The relative performance of adaptive routing may improve if more intelligent mapping heuristics are used. For example, one consequence of achieving deadlock freedom with routing restrictions is that adaptive routing often introduces imbalance into the network by favoring certain message routes. The mapping heuristic may realize better performance by exploiting this imbalance.

The results also suggest that an improvement in the hardware design of adaptive routers is needed to compensate for the added complexity of adaptive routing. Efficient designs of adaptive routers will decrease the difference in router cycle times and should lead to improved performance for adaptive routing. Our simulation results for the hypercube, which use the same router cycle time for both adaptive and nonadaptive routing, support this claim [14].

Finally, additional measures of routing algorithm performance may be required to obtain a clear picture of expected performance under real conditions. For example, if two routing algorithms have similar average message latencies, the one with a lower *variance* seems likely to give better performance in practice [10].

## Acknowledgments

The authors thank Tao Yang for making the DSC and Pyrros source code available and Kalluri Eswar for providing the DAGs used in this paper. We thank Dhableswar Panda for his insights on this problem as well as Jeff May, Kant Patel, and Roshan Rao for developing the simulator. We are indebted to Jeff May for his help with modifications to support the communication between the mapper and the simulator.

## References

- [1] A. A. Chien. A Cost and Speed Model for k-ary n-cube Wormhole Routers. In *Proceedings of the Hot Interconnects Workshop*, August 1993.
- [2] S. Chittor and R. Enbody. Performance Degradation in Large Wormhole-Routed Interprocessor Communication Networks. In *International Conference on Parallel Processing*, volume I, pages 424–428, 1990.
- [3] W. J. Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.
- [4] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.
- [5] K. Eswar, C.-H. Huang, and P. Sadayappan. *Partitioning and Mapping for Parallel Sparse Cholesky Factorization*, 1995. Unpublished Manuscript.
- [6] A. George, M.T. Heath, and J. Liu. Parallel Cholesky Factorization on a Shared Memory Processor. *Linear Algebra and its Applications*, 77:165–187, May 1986.
- [7] C. Glass and L. M. Ni. Maximally Fully Adaptive Routing in 2D Meshes. In *International Conference on Parallel Processing*, volume I, pages 101–104, 1992.
- [8] D. N. Jayasimha, J. May, K. Patel, R. Rao, and L. Schwiebert. *A Simulation Environment for Multiprocessor Architectures*, 1995. Unpublished Manuscript.
- [9] J. May, D. N. Jayasimha, and K. Patel. Comparison of Multiplexing Schemes for Wormhole-Routed Distributed Memory Multiprocessors. In *1<sup>st</sup> International Workshop on Parallel Processing*, pages 211–215, 1994.
- [10] K. C. Patel and D. N. Jayasimha. Measures of Message Latency for Wormhole-Routed Distributed-Memory Multiprocessors. Technical Report OSU-CISRC-4/94-TR23, The Ohio State University, April 1994. Revised April, 1995.
- [11] H. Schwetman. CSIM: A C-Based, Process-Oriented Simulation Language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, 1986.
- [12] L. Schwiebert and D. N. Jayasimha. Mapping to Reduce Contention in Multiprocessor Architectures. In *International Parallel Processing Symposium*, pages 248–253, 1993.
- [13] L. Schwiebert and D. N. Jayasimha. Optimal Fully Adaptive Minimal Wormhole Routing for Meshes. *Journal of Parallel and Distributed Computing*, 27(1):56–70, May 1995.
- [14] L. Schwiebert and D. N. Jayasimha. On Measuring the Performance of Adaptive Wormhole Routing. Technical Report PDCL-97-06-44, Wayne State University – Parallel and Distributed Computing Laboratory, April 1997. Available from <http://www.pdcl.eng.wayne.edu/reports.html>.
- [15] J. Yang, L. Bic, and A. Nicolau. A Mapping Strategy for MIMD Computers. In *International Conference on Parallel Processing*, volume I, pages 102–109, 1991.
- [16] T. Yang and A. Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors. In *ACM International Conference on Supercomputing*, pages 1–10, 1992.
- [17] T. Yang and A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.