A Comprehensive Study of Communication in Distributed-Memory Multiprocessors

By

Loren Schwiebert, Ph.D.

The Ohio State University, 1995

D. N. Jayasimha, Adviser

This dissertation studies the problem of channel contention in distributed-memory multiprocessors. Our main contributions are on adaptive routing. In this dissertation, we pose and answer several fundamental questions related to the optimality and deadlock freedom of routing algorithms. We also address contention through mapping, a complementary software approach.

First, we present a *fully adaptive* routing algorithm for n-dimensional meshes, which allows a message to use any shortest path from the source to the destination. We prove that this routing algorithm is optimal both in its hardware resource requirements and its flexibility in using these resources. Furthermore, we prove that the *only* routing algorithms that can satisfy both optimality conditions are symmetric cases of our routing algorithm.

Building on the ideas underlying these proofs of optimality, we present a necessary and sufficient condition for deadlock-free routing. The resulting proof methodology applies to a much richer class of routing algorithms than previous proof techniques. Furthermore, our methodology is applicable to *any* switching technique. To prove our results, we introduce two new concepts: the *channel waiting graph* and *False Resource Cycles*. These two concepts yield a simple and powerful methodology, which we demonstrate by proposing a *partially adaptive* routing algorithm for wormhole-routed meshes that does not require any additional virtual channels, and fully adaptive routing algorithms for wormhole-routed hypercubes and packet-switched meshes that are more adaptive than any previous routing algorithm for these topologies and switching techniques.

We then present a heuristic for compile-time mapping of clustered task graphs that measures the channel contention in each mapping. This yields two advantages: a more accurate execution time estimate of each mapping and better information for improving the mapping. Simulation results show the advantage of considering contention when mapping.

The mapping heuristic is then integrated with a simulator that models the execution and performance of many routing algorithms, including those that we have proposed. Using task graphs derived from the Cholesky factorization of sparse matrices, we investigate the relationship between mapping and routing. The simulation results demonstrate the advantages of our comprehensive approach to reducing contention.

A Comprehensive Study of Communication in Distributed-Memory Multiprocessors

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the Graduate

School of The Ohio State University

By

Loren Schwiebert, B.S., M.S.

* * * * *

The Ohio State University

1995

Dissertation Committee:

D. N. Jayasimha

Dhabaleswar Panda

Neelam Soundararajan

Approved by

Adviser

Department of Computer and Information Science

To Pam and Joel

Acknowledgments

First and foremost, I thank my adviser, Professor D. N. Jayasimha, for his assistance. When I first started working on my Ph.D., I had no idea how to conduct research. It was only with his patient guidance and thoughtful advice that I developed whatever research capabilities I have exhibited. The research in this dissertation was all produced as a joint effort with my adviser. Without his research input, constructive criticism, and numerous suggestions of problems and extensions, the contributions in this dissertation would have been much less substantial. In addition, his advice on ethical and professional considerations, tips on my job search, etc. has been very much appreciated and valued.

I also thank Professor Dhabaleswar Panda and Professor Neelam Soundararajan for serving on my committee. Their efforts are greatly appreciated and their comments have improved the correctness and clarity of this dissertation.

I thank Professor Panda, Professor Sadayappan, Professor Yalamanchili, and Ms. Eleanor Quinlan for writing recommendation letters for my job search. Furthermore, I thank Professors Panda and Yalamanchili for their recommendation letters for my fellowship application.

I thank the department for supporting me as a teaching associate for five years and the graduate school for awarding me the University Presidential Fellowship. I especially thank Ms. Eleanor Quinlan for her confidence in and support of my teaching. I also thank Carl Phillips and the other computer staff members for their assistance in resolving whatever computer problems I created for myself.

On a technical level, I thank Scott King for his help with using the Unix shared memory facility and Kalluri Eswar for providing the DAGs used in Chapter VI. Bill Pippin and Paolo Bucci were kind enough to allow my simulations to run on their workstations. I thank Manas Mandal for supporting the dissertation style file, which made the preparation of this dissertation significantly easier. I also thank Pete Ware for supporting LaTeXand answering my many questions. Jeff May, Kant Patel, and Roshan Rao wrote the simulator used in Chapter VI. On several occasions, Jeff May and Kant Patel went beyond the call of duty to provide assistance and answer questions. I am more than happy to acknowledge their help.

I have had the opportunity to do research with many graduate students. Specifically, Golden Richard, Frank Adelstein, Mahendra Ramachandran, Raghu Machiraju, Yu-Chee Tseng, Kant Patel, Jeff May, and Roshan Rao. I thank each and every one of them for the opportunity to work with them.

Numerous people have provided feedback on my research at various stages and I appreciate their input. In addition to numerous anonymous referees, I would like to acknowledge Professor Panda, Professor Anish Arora, Yu-Chee Tseng, Shobana Balakrishnan, and Professor José Duato for their suggestions. Furthermore, the students in Professor Jayasimha's research group: Dave Lutz, Jeff Martens, Jeff May, Kant Patel, and Roshan Rao, provided excellent comments and suggestions.

On a personal level, I thank my many friends for their friendship, which has made my days (years) in graduate school immeasurably more enjoyable. I recall many memorable evenings working through homework, hacking C code, and generating graphical images with Frank Adelstein and Golden Richard. I thank Mahendra Ramachandran for his sound advice and frequent help. I thank Kalluri Eswar for his friendship. Mahendra and Eswar deserve special recognition for their willingness to share an office with me for many quarters. I thank my many friends who started graduate school with me and have since left, including Kevin Carpenter, Jim Escoe, Neel Jain, Pierce Krouse, Kent Lathrop, and Shantanu Sarkar. I am grateful for the friendship of Sandeep Gupta and Shivnandan Kaushik and for their demonstration of the amount of time and dedication required to complete the dissertation. I thank my recent office mates, especially Paolo Bucci, Gopal Dommety, and Bill Pippin for their friendship and help during these last few months. To my many other friends, I thank you and regret that I am unable to mention everyone by name.

I thank Neil Lehman, my wife's uncle, for invaluable assistance and advice on pursuing an academic degree and career, and my in-laws, Dewey and Kathy Morrow, for their emotional and financial support. My grandparents, B. Keith and Isabella Pyle and Albert "Bert" and Lucille Schwiebert, have been a constant source of support and inspiration. I thank my parents, Loren and Janice Schwiebert, for their emotional support and their loving care during my formative years. I thank my wife, Pam, for her unwavering confidence and support throughout this entire process, but especially during the final few months. At times, it was only her confidence that enabled me to persevere. These few sentences cannot adequately express my love and appreciation of her. Last, but not least, I thank my son, Joel, for all the joy that he has brought to my life. I also thank him for constantly reminding me of what is truly important in life.

Vita

June 5, 1964	Born - Springfield, IL
1986	B.S. Computer Science Heidelberg College Tiffin, Ohio
1990	M.S. Computer and Information Science The Ohio State University Columbus, Ohio
1989-1994	Graduate Teaching Associate, The Ohio State University.
1994-1995	University Presidential Fellow, The Ohio State University.

Publications

Loren Schwiebert and D. N. Jayasimha. A Necessary and Sufficient Condition for Deadlock-Free Wormhole Routing. To appear in *Journal of Parallel and Distributed Computing.*

Loren Schwiebert and D. N. Jayasimha. A Universal Proof Technique for Deadlock-Free Routing in Interconnection Networks. To appear in 7^{th} *Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1995.

Loren Schwiebert and D. N. Jayasimha. Optimal Fully Adaptive Minimal Wormhole Routing for Meshes. To appear in *Journal of Parallel and Distributed Computing.*

Frank Adelstein, Golden Richard, Loren Schwiebert, Rick Parent, and Mukesh Singhal. A Distributed Graphics Library System. *Software—Practice and Experience*, 24(4):363–376, April 1994.

Loren Schwiebert and D. N. Jayasimha. Optimal Fully Adaptive Wormhole Routing for Meshes. In *Supercomputing '93*, pages 782–791, November 1993.

Loren Schwiebert and D. N. Jayasimha. Mapping to Reduce Contention in Multiprocessor Architectures. In *IEEE* 7^{th} *International Parallel Processing Symposium*, pages 248–253, April 1993.

Loren Schwiebert and D. N. Jayasimha. A Necessary and Sufficient Condition for Deadlock-Free Wormhole Routing. Technical Report OSU-CISRC-4/94-TR22, April 1994 (revised December 1994).

Loren Schwiebert and D. N. Jayasimha. Optimal Fully Adaptive Wormhole Routing for Meshes. Technical Report OSU-CISRC-4/93-TR16, April 1993 (revised August 1993).

Raghu Machiraju, Loren Schwiebert, and Roni Yagel. Parallel Algorithms for Volume Rendering. Technical Report OSU-CISRC-10/92-TR29, October 1992.

Loren Schwiebert and D. N. Jayasimha. Mapping Parallel Computations to Multiprocessor Architectures Considering the Effects of Communication. Technical Report OSU-CISRC-5/92-TR14, May 1992.

Fields of Study

Major Field: Computer and Information Science

Studies in:

Parallel Computing	Prof. D. N. Jayasimha
Computer Architecture	Prof. Dhabaleswar Panda
Computer Graphics	Prof. Rick Parent, Prof. Roni Yagel

Table of Contents

List of Tables

List of Figures

CHAPTER I

Introduction

Parallel processing is the area of computer science that proposes and studies methods of improving the performance of computer systems by using more than one processor concurrently to solve a problem. When the processors are tightly-coupled, such computers are typically referred to as either parallel computers, parallel processors, or multiprocessors. The processors are used in parallel, with each processor computing a different part of the parallel program. In this way, the execution time of the program is reduced. Alternatively, a multiprocessor can be used to solve larger instances of a problem than would be feasible on a uniprocessor.

When a program is partitioned among the processors, communication is required to share the results of computations among processors that need access to these results. In addition, processors typically need to synchronize at various stages of the computation before proceeding. This ensures, for example, that results from one iteration or phase of the computation are not overwritten before the results have been used by other processors.

Communication overhead is widely recognized as an obstacle to achieving scalable performance on multiprocessors. For this reason, many researchers have proposed techniques for reducing this overhead. As discussed later, reducing contention for channels in the interconnection network is of increasing importance in reducing communication overhead. This dissertation studies the problem of channel contention in distributed-memory multiprocessors. Our main contributions are on adaptive routing, an architectural approach to reducing contention. In this dissertation, we pose and answer several fundamental questions related to the optimality and deadlock freedom of adaptive routing algorithms. We also address contention through mapping, a complementary software approach. The primary contributions in this dissertation can be divided into three categories:

1. An optimal fully adaptive routing algorithm for wormhole routing on arbitrary dimension meshes.

2. A necessary and sufficient condition for deadlock-free routing.

3. A mapping heuristic that focuses on reducing channel contention. The channel contention is also considered when evaluating mappings and selecting the best mapping from the set of generated mappings.

To put these results in context, an introduction to architectural and software issues is given in this chapter. In Chapter II previous work on adaptive routing and mapping is discussed. Chapter II concludes with an outline of the remainder of the thesis.

1.1 Multiprocessor Architectures

There are two possibilities when designing a multiprocessor. A multiprocessor can have either shared memory or distributed memory. Regardless of the classification, an interconnection network is used to logically connect the processors. Figure 1(a) depicts a simple shared-memory multiprocessor architecture. There are k global memory modules, denoted, GM. There are also n processing elements, denoted PE, where each PE has local memory and a processor. The PE is connected to the interconnection network through the a network interface, denoted NI. The interconnection network allows each processing element access to the global memory modules. Thus, the interconnection network functions analogously to a memory bus [49]. Figure 1(b) depicts a simple distributed-memory multiprocessor. Since a larger local memory is assumed, the processor, denoted Proc, and the local memory, denoted LM, are shown separately. The interconnection network is used to transmit messages between processing elements, rather than from a processor to a global memory module. Hence, for distributed-memory multiprocessors, the interconnection network functions like an I/O interface [49].

The design in Figure 1 can be extended by introducing a two-level interconnection network, where the processing elements designate clusters of processors instead of individual processors. The processors in each cluster then use an intra-cluster network for intra-cluster communication and use the interconnection network only for inter-cluster communication. This two-level hierarchy can in turn be generalized to an arbitrary number of levels.

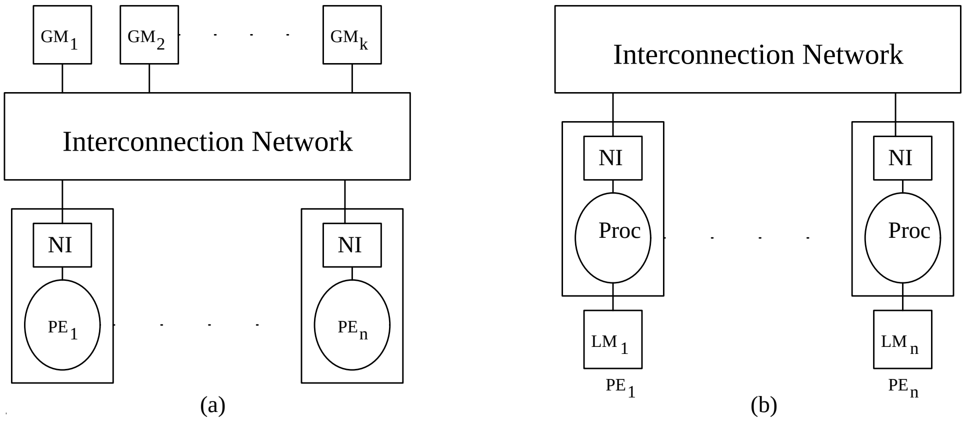


Figure 1: Shared-Memory and Distributed-Memory Architectures

Independent of the architectural model, the programming model can use either a shared-memory paradigm or a message-passing paradigm. The shared-memory paradigm is generally easier from a programming standpoint, so the use of message passing to emulate distributed memory on a shared-memory multiprocessor is uncommon. The shared-memory model on a distributed-memory multiprocessor is supported by distributed-shared memory. The different programming models are discussed further when the advantages and disadvantages of shared-memory and distributed-memory multiprocessors are described.

If a parallel application is inherently parallel, then a multiprocessor should achieve nearly linear speedup as the number of processors is increased. Such a multiprocessor architecture is called *scalable*. Building scalable shared-memory multiprocessors is difficult, because access to the global memory modules quickly becomes a bottleneck. A processor writes results to a global memory address before other processors can read these results. Another processor can then read this result from global memory.

Explicit synchronization is required to prevent processors from reading results before they are written and from overwriting results before they are read. Synchronization is often implemented with busy waiting, which requires a processor to repeatedly access a global memory address until the read or write is successful.

As the number of processors is increased, the memory bandwidth should also increase. This is difficult, since a bus, or an indirect network such as an Omega network, provides access to the global memory. The number of stages in the network (or the size of the bus) increases with the number of processors. Thus, even if sufficient memory bandwidth can be provided, the minimum network latency increases with the number of processors. Even though techniques such as message combining [81] and hierarchical shared memory [46] have been proposed for reducing the message traffic and message latency, most existing large-scale multiprocessors use distributed memory.

Distributed-memory multiprocessors are considered to be more scalable. The message latency on a distributed-memory multiprocessor is partially dependent on the distance between the two processors, where distance is measured as the number of network channels that must be traversed to reach the destination. The distance a message travels is also referred to as the number of "hops" from the source to the destination. As the number of processors increases, the network diameter, also increases. There is non-uniform latency, however, because all processors are not equidistant. If the program is partitioned in a judicious manner, a larger fraction of the messages either can be eliminated by placing communicating tasks on the same processor or can be sent between processors that are relatively close. In addition, a great deal of research has focused on making the message latency less dependent on the distance between processors. Furthermore, the transmission and reception of messages enforces implicit synchronization among the processors, so network bandwidth is not consumed by busy waiting. These approaches reduce the effect an increased network diameter has on the message latency, although contention for network channels should also be addressed.

The main drawback to distributed-memory multiprocessors is the difficulty of writing efficient parallel programs. In order to support communication on a distributed-memory multiprocessor, the message-passing model requires the program code to designate both the sending and receiving processors for each message. This increases the complexity of programming on distributed-memory multiprocessors. Distributed-shared memory has been proposed to address this disadvantage. In this programming paradigm, the memory is physically distributed but logically shared. The programmer can assume a global address space and use the multiprocessor as a shared-memory multiprocessor. The operating system converts memory requests for non-local data into messages. With distributed-shared memory, communication is receiver-initiated; a processor sends a request for data whenever the operating system determines that a non-local memory access is required. This differs from the message-passing paradigm, which requires sender-initiated communication. Although from the programming standpoint distributed-shared memory simplifies the task of writing parallel programs, the issue of efficient communication remains.

Because of the need for scalability, our research has focused on distributed-memory multiprocessors. There are several factors that affect the scalability of a multiprocessor. Distributed-memory multiprocessors rely on efficient interconnection network communication to achieve scalability. The partitioning of parallel programs also affects the performance of a distributed-memory multiprocessor. The need for reasonable partitioning increases as the number of processors increases, since the average distance between processors increases.

The network topology defines the connectivity of the interconnection network. A network topology with more channels can reduce the diameter and average distance of the network, however, a completely connected network is not practical. For example, consider a relatively modest multiprocessor consisting of 1024 processors. A completely connected network would require over one million network channels. The pin-out constraints alone make such a network infeasible, since each processor would require 1023 channels. Even for smaller networks, the bandwidth of each channel would be severely restricted. Assuming networks with uniform bisection bandwidth, Dally [14] has shown that interconnection networks with a few wide channels can sustain higher throughput and achieve lower latency than networks with many narrow channels.

1.2 Network Topologies

Three topologies are commonly used in current multiprocessors: mesh, torus, and hypercube. As discussed later, such topologies are often referred to as k-ary n-cubes.

Other topologies have been proposed, such as de Bruijn networks [85] and cube-connected cycles [84]. As far as we know, no commercial multiprocessors use these topologies.

1.2.1 Hypercube

A hypercube has two processing nodes in each dimension, so a hypercube can be characterized as a 2-ary n-cube, where n is the dimension. For example, a 2-ary 8-cube is an 8D hypercube. An n-dimensional hypercube consists of 2^n nodes, where each node can be uniquely labeled using an n bit string. For example, node $H = (h_{n-1}h_{n-2}\ldots h_1h_0)$. In a hypercube, there is a channel between nodes S and D if the labels for S and D differ in exactly one bit. Otherwise, a message from S to D needs to be routed through intermediate nodes that move the message closer to D. The diameter of an nD hypercube is n and each processor has n channels, one for each of its neighboring nodes. An example of a 3D hypercube is shown in Figure 2.

1.2.2 Torus

A torus can be characterized as a k-ary n-cube, where k is the radix. Unlike a hypercube, a torus can have more than two nodes per dimension. For example, a 16×16 torus is a 16-ary 2-cube. A k-ary n-cube has the same number of nodes in each dimension, however, a torus can be generalized so that the number of nodes in each dimension is not uniform. Hence, an 8×5 torus is a torus with $k_0 = 8$ and $k_1 = 5$. Each node in an n-dimensional torus can be represented by an n-tuple, $T = (t_0, t_1, \ldots, t_{n-1})$, where $\forall i:\ 0 \leq t_i < k_i$. Two nodes on a torus, S and D, are

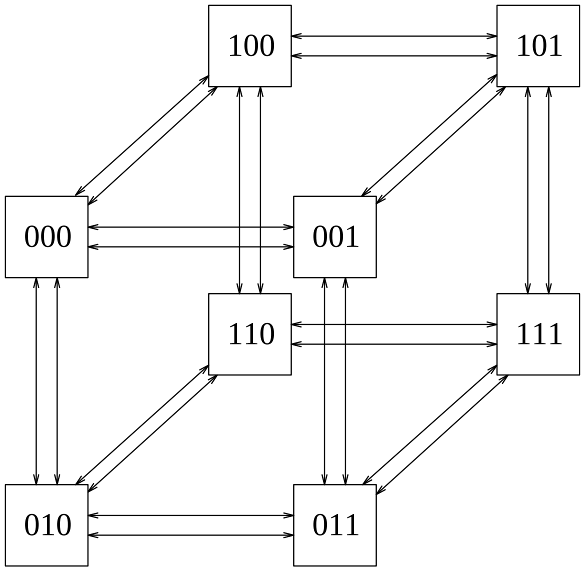


Figure 2: A 3D Hypercube

neighbors if the nodes differ in only one dimension, j, and $s_j = d_j \pm 1 \pmod{k_j}$. Accordingly, a 16×16 torus has a network channel, called a wrap-around channel, between nodes (3,0) and (3,15). An n-dimensional torus has $k_0 \times k_1 \times \ldots \times k_{n-1}$ nodes and $2n$ channels per node. The diameter of an n-dimensional torus is $\sum_{i=0}^{n-1} \lfloor \frac{k_i}{2} \rfloor$. An example of a 4×4 torus is shown in Figure 3.

1.2.3 Mesh

A mesh is similar to a torus, except for the absence of wrap-around channels. Although sometimes referred to as a k-ary n-mesh, there is no requirement that a mesh have the same number of nodes in each dimension. Each node in an n-dimensional mesh can be represented by an n-tuple, $M = (m_0, m_1, \ldots, m_{n-1})$, where $\forall i : \ 0 \leq m_i < k_i$. Two nodes on a mesh, S and D, are neighbors if the nodes differ

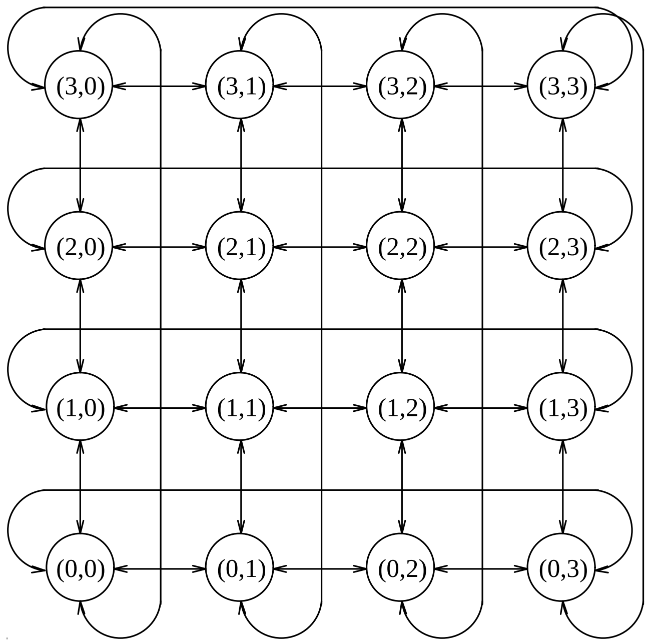


Figure 3: A 4×4 Torus

in only one dimension, j, and $s_j = d_j \pm 1$. Hence, there are no wrap-around channels. An n-dimensional mesh has $k_0 \times k_1 \times \ldots \times k_{n-1}$ nodes. Nodes on the interior of the mesh have $2n$ channels, while nodes on the periphery have between n and $2n-1$ channels. The diameter of an n-dimensional mesh is $\sum_{i=0}^{n-1}(k_i - 1)$.

1.3 Switching Techniques

An interconnection network is logically connected. Otherwise, some pairs of nodes would be unable to communicate. Since it is not feasible to provide a channel between every pair of nodes, the network channels are shared among the processing nodes. For some source-destination pairs to communicate, messages must traverse routers at intermediate nodes along a path from the source to the destination.

The switching technique defines how messages are propagated through the network. More precisely, the switching technique defines the hardware and software protocols for transmitting and buffering data when sending a message between neighboring routers. A variety of switching techniques have been proposed for supporting communication across the multiple network channels between the source and the destination. The most common techniques, including circuit switching, store-and-forward routing, virtual cut-through, and wormhole routing are described in detail. Other switching techniques, such as adaptive cut-through and pipelined circuit switching are briefly described.

For the most common switching techniques, we derive lower bound expressions on message latency between two nodes of the distributed-memory multiprocessor using the following parameters. The message latency is given in network clock cycles, denoted by a parameter, t_i, where i changes to indicate that the cycle times may be different with different switching techniques. The message length is denoted L_m. The length of the message header is denoted L_h and the length of the circuit acknowledgment is denoted L_a. The formula also uses the number of hops, h, and the width of the network channels, W. The start-up cost of the message is denoted L_s.

1.3.1 Circuit Switching

Circuit switching first sends a probe from the source to the destination and returns an acknowledgment once the path is established. If the probe encounters a busy channel, the partial path that has been established is aborted and the probe is retried later. Alternatively, a time-out mechanism can be used so that the probe waits for a

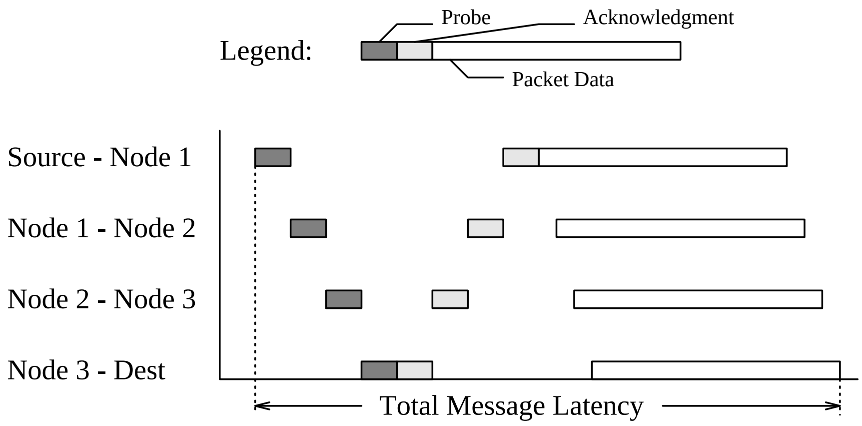


Figure 4: Minimum Latency for Circuit Switching

finite period of time before aborting. After the acknowledgment is received, multiple messages can be transmitted on this path before the path is released.

There are a few advantages to circuit switching. For relatively long messages, the message latency is almost independent of the distance between the source and destination. In addition, there are no storage requirements at the routers, since no data is sent until the circuit has been established. The only storage required is a modest amount, at most a few bytes, to temporarily hold a probe. The minimum latency for a message that travels four hops through the network is shown in Figure 4. The formula for the minimum message latency, L, in network clock cycles is:

$$L = (L_s + h(L_h/W) + h(L_a/W) + L_m/W + h)t_c \tag{1.1}$$

The main disadvantage of circuit switching is the blocking that each circuit causes. Since a circuit requires exclusive use of all the channels on a path, and the path can be held for a relatively long period of time after being established, many probes can be blocked by a single circuit. Even short messages can reserve many channels and every channel is held for the duration of the circuit. Although multiple messages can be transmitted once the circuit is established, this increases the length of time a circuit is held and increases contention for network channels. Another disadvantage of circuit switching is the time required to establish a circuit. Since the probe is transmitted to the destination and then acknowledged before any data is sent, the time to establish the path can dominate the time required to send the message, especially when the diameter of the network is relatively high and the messages are relatively short. Finally, having to retry blocked probes can waste network bandwidth and reduce the network throughput.

1.3.2 Store-and-forward Routing

As opposed to circuit switching, *store-and-forward* routing sends the message before the entire path has been established. Store-and-forward routing is implemented via packet switching or message switching. Packet switching packetizes each message and sends each packet individually, whereas message switching sends the entire message as a single unit. The network treats each packet as a separate message, so from the perspective of the network, the only difference between the two is that packet switching has messages of fixed length. For this reason, we use the terms message and packet interchangeably. Packet switching transmits the packet header and the all the packet data immediately to a neighboring node. The entire packet is then stored in a buffer at this node and later forwarded to the next node in the path.

When transmitting a packet between neighboring nodes, the packet is briefly stored in buffers on both nodes before the buffer on the previous node is released. Most of the time, however, the packet occupies a single buffer. Hence, each packet in the network is holding relatively few resources and there is less potential for blocking among the packets.

Since the entire packet is buffered at each intermediate node before any part of the packet is forwarded, the message latency is the product of the packet length and the distance between the source and destination. This is acceptable for small packets on networks with a relatively small diameter, however, the message latency for long packets or small packets on a network with a relatively large diameter can be unacceptable. The minimum latency for a message that travels four hops through the network is shown in Figure 5. The formula for the minimum message latency, L, when a message is a single packet is:

$$L = (h(L_s + (L_h + L_m)/W))t_p \tag{1.2}$$

The packet buffers are usually too large to be placed on the router, so the packet is transferred to the processing node and stored in system memory at each intermediate router. This increases the message latency, because the packet incurs setup overhead at each node. Furthermore, each intermediate node has to provide system memory in which to store the packet and bandwidth between the processor and the

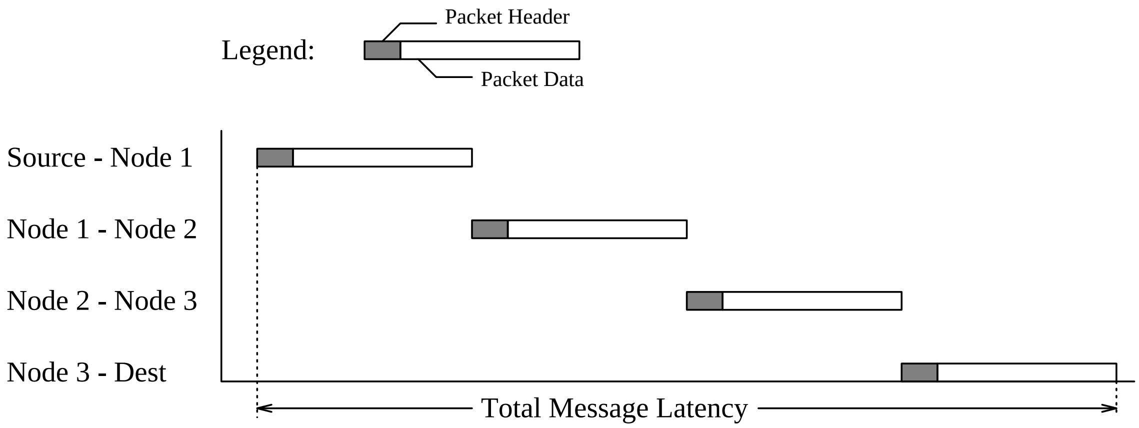


Figure 5: Minimum Latency for Store-and-forward Routing

interconnection network is wasted. Circuit switching, on the other hand, does not require any interaction between the routers and processing elements on the intermediate processors.

1.3.3 Virtual Cut-through

Kermani and Kleinrock [52] proposed *virtual cut-through* to combine the best attributes of packet switching and circuit switching. Virtual cut-through works like packet switching with one important difference. When the packet header arrives at an intermediate node and the next buffer in the path is available, the packet header is routed immediately to the next node. This prevents the unnecessary transfer of packets to the packet buffers. It is now possible that the contents of the packet span several routers. This allows the data to be sent in a pipelined fashion across the network channels. Under ideal conditions, the message latency is less than circuit switching, since circuit switching has the additional overhead of sending and

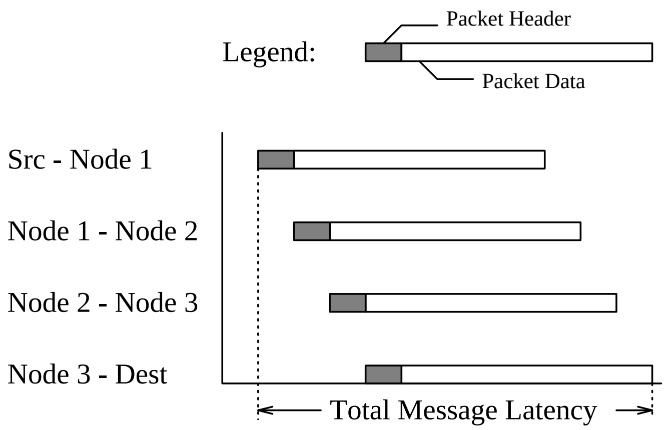


Figure 6: Minimum Latency for Virtual Cut-through

acknowledging the probe before any data is sent. With virtual cut-through, the first data element arrives at the destination immediately after the packet header reaches the destination. Thus, the minimum latency is the sum of the packet length and the distance between the source and the destination. The minimum latency for a message that travels four hops through the network is shown in Figure 6. The formula for the minimum message latency, L, is:

$$L = (L_s + h(L_h/W) + L_m/W)t_v \tag{1.3}$$

A packet is buffered only when the next buffer is not available. This reduces the resources that a blocked packet holds. Although a packet can temporarily occupy many channels, those channels are being used productively, since data is being sent across them. When a packet is blocked, the data continues along the channels until it reaches the router where the message was blocked. The data then leaves the network and is buffered in system memory. Conversely, circuit switching holds all the channels along the path from the source to the destination. Even when a probe is unsuccessful, many channels can be held temporarily while the probe is attempting to establish a path.

Since blocked packets are buffered in system memory on the processing node, there is still the need to transfer packets out of the network and later reinject the packets. For lightly and moderately loaded networks, however, the packets are buffered less often with virtual cut-through than with store-and-forward routing.

1.3.4 Wormhole Routing

Dally and Seitz [17] proposed *wormhole routing* to avoid the large buffers required for virtual cut-through and store-and-forward routing. Wormhole routing logically partitions each packet into flow control units (flits). A packet consists of one or more header flits, the data flits, and, optionally, a tail flit to mark the end of the packet. The length of a flit corresponds to the width of a network channel and is usually no more than a couple of bytes. Since flits do not contain any header information, flits from different packets cannot be interleaved in the same buffer. Rather than providing large buffers for a packet, a small buffer is associated with each channel in the network. This buffer is typically large enough to hold only a few flits.

Until a packet is blocked, wormhole routing transmits packets in the same way as virtual cut-through. When packets are blocked, however, the packets remain in the network. By eliminating the need for large buffers, the problem of having to remove a packet from the network at intermediate nodes is also eliminated. An additional benefit is that the network cycle time can be decreased. Since the buffers are small relative to the length of a packet, however, blocked packets typically occupy the buffers of all the channels used from the source to the intermediate node where the packet is blocked.

Jesshope, Miller, and Yantchev [48] proposed *mad postman* routing as a more efficient implementation of wormhole routing. Mad postman routing overlaps the forwarding of the message header with the routing decision of selecting the output channel for the message. If the chosen output channel is available and needed by the message, the overhead of determining the routing path is avoided at that node.

Under ideal conditions, *i.e.*, low network traffic, the latency is the same for wormhole routing and virtual cut-through, except for a change in the network cycle time. With higher network traffic, however, wormhole routing is susceptible to some of the same problems that are experienced with circuit switching, because a message holds a channel from the time the header acquires the channel until the tail releases the channel. Unlike circuit switching, however, wormhole routing begins sending data as soon as the packet enters the network. For this reason, terminating blocked packets is not possible. Hence, wormhole routing does not have the drawback of unsuccessful probes wasting channel bandwidth.

The primary drawback to wormhole routing is the contention that can occur with even moderate network traffic, which results in higher message latency. Whenever a message is unable to proceed due to contention, the packet holds all the channels it currently occupies. Since all the channels in the path from the source to the destination are held from the time they are acquired until the entire packet has traversed the channel (which is after the entire path has been established except for relatively short messages that fit in the intermediate channel buffers), performance degradation due to contention can be severe and message latency can be unacceptably high. A message that uses several channels can block many messages while being transmitted. These blocked messages can in turn block other messages, which further increases the message latency and reduces the throughput. Providing additional *physical* channels between neighboring nodes can reduce both latency and contention. This is an expensive solution, however, and can actually increase message latency if the routers are pin-out constrained. A more cost-effective method of reducing message latency, proposed by Dally [15], is to allow multiple *virtual* channels to share a physical channel. Each virtual channel has a separate buffer, with multiple messages multiplexed over the physical channel. Virtual channels reduce contention, because more channels are available and messages can route around blocked messages. If time multiplexing is used to share these channels, then every virtual channel is guaranteed a fixed amount of channel bandwidth, even when the virtual channel is empty or is occupied by a blocked message. Demand multiplexing allows the channel bandwidth to be shared among only those virtual channels that are able to transmit messages. Thus, blocked messages do not consume channel bandwidth. Ni and McKinley [77] offer an in-depth discussion of wormhole routing.

1.3.5 Adaptive Cut-through

Ngai and Seitz [76] proposed *adaptive cut-through* as a modification of virtual cut-through. The only difference is that blocked packets that cannot be routed toward the destination are routed away from the destination (misrouted) instead of buffered. Since each router has the same number of input channels and output channels, every packet can always be routed. Hence, packets are never buffered. A disadvantage of adaptive cut-through is that one of the input channels to the network is from the processing node. To keep the number of input and output channels balanced, it may be necessary to transmit messages to the processing node even when this node is not the destination of the packet. This consumes processor-network bandwidth, however, the additional storage requirements incurred are modest, since at most one packet at a time can be misrouted to the processor. Because packets can be misrouted, it is difficult to guarantee that packets eventually reach their destinations. This is overcome by introducing a priority scheme, however, the overhead of transmitting the extra information required for maintaining priorities consumes additional channel bandwidth. The minimum message latency for adaptive cut-through is the same as virtual cut-through, except for the additional header length and the possible difference in network cycle time.

1.3.6 Pipelined Circuit Switching

Gaughan and Yalamanchili [33] proposed *pipelined circuit switching* primarily to support fault tolerance. Each physical channel is partitioned into several virtual channels, which are multiplexed over the physical channel. As previously mentioned, it is not possible to abort a blocked packet when wormhole routing is used, because data from the packet has already entered the network. This limitation does not exist with circuit switching, since no data enters the network until the probe establishes the circuit and an acknowledgment is received. If a probe is unable to establish a circuit because of faulty network channels, the probe can be aborted and retried at a later time. Circuit switching requires exclusive use of network channels, however, the virtual channels used with pipelined circuit switching allow the network channels to be shared. Hence, although the total bandwidth is the same, the channel contention should be lower. The minimum message latency for pipelined circuit switching is the same as circuit switching.

1.4 Network Routing

The switching technique determines how messages are propagated from the source to the destination, including the hardware protocols for transmitting data across a physical channel and for buffering data at a router. The routing algorithm, on the other hand, determines the set of paths between a source and destination. The path selection can have a significant effect on the message latency. Thus, a more flexible routing algorithm could improve the network throughput and reduce the average message latency.

A routing algorithm is comprised of two parts: a routing relation and an output selection function. The routing relation defines the set of output channels a message can use. This determination is based on parameters such as the destination of the message and the input channel used by the message. The output selection function chooses a single channel from this set, based on the availability of the channels. The message is then forwarded along this channel. An input selection function is also required to route messages. The input selection function determines the priority of messages simultaneously arriving at the same router. The input selection function is independent of the routing algorithm. Hence, when routing algorithms are discussed, the term selection function refers to the output selection function.

When designing routing algorithms, the network performance is an important consideration. It is not, however, the only issue to consider. The hardware resources required by a routing algorithm should be reasonable, so that the router design is cost-effective. In addition, routing should be deadlock-free and livelock-free, and should not be prone to starvation.

In an interconnection network, deadlock can arise whenever it is possible for a set of messages to permanently block the progress of each other. The network is never able to deliver any of these messages, since the progress of each message depends on progress by other messages in the set. (Deadlock could also arise when a single message prevents itself from reaching its destination.) Since a parallel program relies on the delivery of all messages in order to complete execution, the routing algorithm must be deadlock-free. Switching techniques such as store-and-forward routing, virtual cut-through, and wormhole routing rely on routing restrictions to avoid deadlock. Other methods of avoiding deadlock, such as the abort/retry mechanism used in circuit switching, and misrouting used in adaptive cut-through, are inherently deadlock-free.

An example of a deadlock configuration with wormhole routing is shown in Figure 7. In this example, we assume messages can use only shortest paths from the

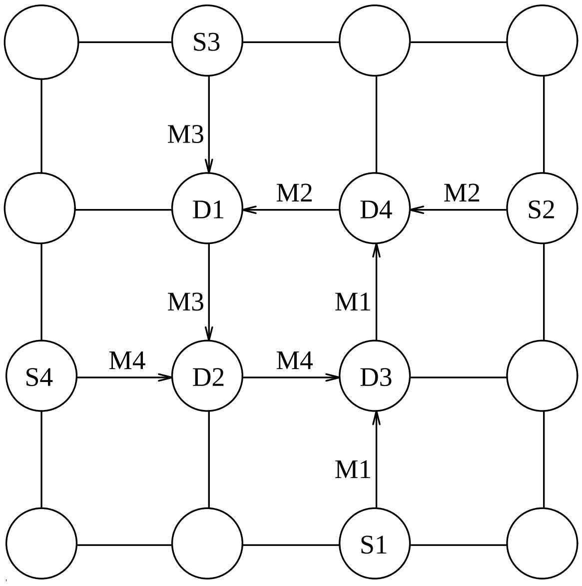


Figure 7: A Deadlock with Wormhole Routing

current node to the destination. The routers are represented as circles with the channels as edges connecting the routers. The source and destination routers of message i are denoted Si and Di, respectively. The channels are labeled Mi to indicate that message i has acquired that channel. Since the channels are bidirectional, arrowheads are used to indicate the direction in which messages are proceeding.

Livelock in an interconnection network occurs whenever a message is continually routed in the network, yet never reaches the destination. The difference between deadlock and livelock is that the messages in a deadlock configuration are blocked. On the other hand, messages in a livelock configuration are able to route, but are unable to reach their respective destinations. As with deadlock, the message is never delivered, so any routing algorithm should be livelock-free. For some switching techniques, such as circuit switching, livelock can occur from continually retrying a probe that can never establish a path. Even though the message is not continually routed through the network, the probe is continually tried and retried without success.

A third condition to be avoided is starvation. Starvation occurs when the allocation of network resources is not fair. In the absence of fairness, a message could be denied a channel, although the channel is continually available and is given to other messages. Obviously, starvation prevents delivery of a message. Avoiding starvation depends on the input selection function, hence, it is independent of the routing algorithm. Starvation can be avoided by allocating buffers and channels in a fair manner, such as in a FIFO or a round robin order.

The simplest routing algorithms are *nonadaptive* and define a single path between the source and destination. Nonadaptive routing algorithms have also been referred to as *deterministic*, because the path taken by a message is predetermined, and *oblivious*, because the path taken by a message is oblivious to the current network traffic. We prefer the term nonadaptive, because the meaning is clear and unambiguous.

In general, it is quite easy to prove that nonadaptive routing algorithms are deadlock-free and livelock-free. Furthermore, nonadaptive routing algorithms can be implemented in hardware with relative ease. Despite these advantages, nonadaptive routing algorithms make inefficient use of the network. Blocked messages are unable to proceed along alternative paths, even when those paths are available.

Adaptive routing algorithms, on the other hand, support multiple paths between the source and destination. Latency and contention can be reduced by using these multiple paths. When a message is unable to proceed along one path, an alternative path can be chosen. This permits more efficient use of the network. Although a choice of paths is permitted, the decision of which path to choose is made using only local information. Local information is limited to the status of the output channels, the location of the current router relative to the destination of the message, and the information contained in the message header. In this sense, routing can be viewed as a greedy algorithm, where the chosen output channel is locally optimal but may not be globally optimal. For example, it is possible that waiting for a busy channel at an intermediate router results in lower overall latency than using a free channel at this router. Although using global information may appear to be advantageous, the overhead of accumulating this information and incorporating this information into the routing algorithm is higher than the benefit. Furthermore, it is not even clear how this information can be incorporated to achieve higher throughput. Despite the inaccuracy of using only local information, adaptive routing still usually outperforms nonadaptive routing, because the network channels are used more efficiently [9, 25, 39].

Routing algorithms are either minimal or nonminimal. Minimal routing algorithms allow only shortest paths to be chosen, while nonminimal routing algorithms do not require messages to use only shortest paths. With minimal routing, every routing step moves the message closer to the destination. Thus, the maximum number of routing steps for any message is the diameter of the network and livelock is impossible. Livelock may arise with nonminimal routing, however, because messages are permitted to route away from the destination. Livelock freedom is guaranteed by restricting the situations that permit nonminimal routing.

Minimal routing has the advantage that every message uses only the minimum number of channels required to reach its destination. As the message traffic increases, the use of extra channels can increase the message latency. Nevertheless, nonminimal routing may improve the latency when attempting to route around hot spots in the network. In addition, nonminimal routing is necessary for fault-tolerance, since a faulty channel between two nodes prevents communication between them unless a nonminimal path around the fault is permitted. There are a variety of choices for minimal and nonminimal routing. For example, a nonminimal routing algorithm may or may not permit a message to backtrack along the path it has already taken. Gaughan and Yalamanchili [34] present a comprehensive overview of the different adaptive routing protocols that exist.

Whether minimal or nonminimal, adaptive routing algorithms can be further differentiated by the fraction of *shortest* paths they allow. *Partially adaptive* routing algorithms do not allow *all* packets to use *any* shortest path. *Fully adaptive* routing algorithms *do* allow all packets to use any shortest path. In other words, at the source and every intermediate router on a path from the source to the destination, a fully adaptive routing algorithm allows a packet to route to any neighboring node that is closer to the destination. Partially adaptive routing algorithms usually do not reduce message latency as much as fully adaptive routing algorithms do. Partially adaptive routing algorithms, however, can be significantly more efficient than nonadaptive routing algorithms. In addition, fully adaptive routing algorithms usually require extra buffers[1] in order to guarantee deadlock freedom. Although all fully adaptive routing algorithms allow a message to use any shortest path through the network, different restrictions may be placed on the choice of buffers on that path. Hence, not all fully adaptive routing algorithms are equivalent. Some fully adaptive routing algorithms allow more adaptiveness than others by placing fewer restrictions on the choice of buffers.

The *degree of adaptiveness* of a routing algorithm can be defined precisely. Although various definitions have been proposed, the most popular is the measure defined by Glass and Ni [39]. The degree of adaptiveness is the ratio of the number of paths permitted by the routing algorithm, to the total number of paths, averaged over all source-destination pairs. For a network with virtual channels, each virtual channel on a physical channel is considered separately and two paths are considered distinct if at least one virtual channel is different.

With wormhole routing, the number of virtual channels required by a routing algorithm gives a good approximation of the router complexity, because each virtual channel needs a separate buffer and the virtual channels are multiplexed over the physical channel. Routing algorithms that require more virtual channels need additional router control logic and are usually more complex. Multiplexing and scheduling virtual channels on a physical channel is more complicated with additional virtual channels. Router latency and network cycle time also increase with the number of virtual

[1]In wormhole routing, a virtual channel is implemented by adding an additional input and output buffer to the physical channel.

channels [8], so fewer virtual channels are usually better. In general, the less adaptive a routing algorithm, the more modest the virtual channel requirements, however, this is not always the case. For example, some fully adaptive routing algorithms [90] require fewer virtual channels than some partially adaptive routing algorithms [9].

Reducing the number of virtual channels needed for a given degree of adaptiveness is accomplished by using a less restrictive routing algorithm. Conversely, when the same number of virtual channels is used, a less restrictive routing algorithm has better performance than a more restrictive routing algorithm [38, 73].

1.5 Mapping

Another factor that affects the efficiency of parallel programs is how the parallel program is partitioned among the processors. The actual assignment of the parallel program to the processors is called *mapping*. Once the mapping has been done, the source and destination processors for each message are known and the communication cost can be determined. If the source and the destination are on the same processor, then no inter-processor communication is required and a message is eliminated. Otherwise, the message must be transmitted from the source to the destination.

Researchers have used a variety of definitions of mapping. Because our interest is in improving the efficiency of distributed-memory multiprocessors, we restrict our focus to the problem of scheduling task graphs with communication costs on a parallel architecture. The approaches to mapping can be divided between compile-time (static) and run-time (dynamic) methods. The disadvantages of run-time mapping are the reduced efficiency that results from making decisions without global information and the overhead of simultaneously mapping and executing the algorithm. These disadvantages are avoided by using compile-time mapping. The drawback to compile-time mapping is the difficulty of determining the execution time for branches and loops. Compile-time mapping is investigated in this dissertation, so our discussion is restricted to compile-time mapping algorithms.

Formally, the parallel application is modeled by a directed acyclic graph (DAG), also called the task graph. The DAG is defined as a graph, $G = (V, E, C, T)$ where each $v_i \in V$ is a task and each $t_i \in T$ denotes the computation cost of v_i. Each edge $e_{i,j} \in E$ is a directed edge from v_i to v_j, with the communication cost, $c_{i,j} \in C$, denoting the amount of data transmitted on edge $e_{i,j}$ if nodes v_i and v_j are executed on distinct processors.

The number of tasks depends solely on the characteristics of the parallel algorithm and is independent of the number of processors in the multiprocessor. Since the task graph is determined at compile-time, each task has a fixed computation cost and each edge has a fixed communication cost. The tasks and edges have individual costs which may be independent of each other, *i.e.*, there is no requirement that all tasks have the same computation cost or all edges have the same communication cost. The edges that connect the tasks not only indicate the need to transmit data between the tasks, but also enforce a partial ordering on the execution of the tasks.

The granularity of a task graph is defined by the ratio of computation to communication. Yang and Gerasoulis [103] define a DAG as coarse-grained if the communication cost of every edge is not more than the computation cost of the source and destination tasks. Formally, $\forall e_{i,j} \in E,\ c_{i,j} \leq t_i \wedge c_{i,j} \leq t_j$. Otherwise, the DAG is a fine-grain DAG.

A variety of measures have been proposed for evaluating the quality of a mapping. It is possible to ignore the communication cost, but this is not acceptable for distributed-memory multiprocessors, because communication can have a significant effect on the execution time of parallel programs. Minimizing the communication time is also not acceptable, since this is achieved by placing all the tasks on the same processor. Ideally, one would like to map the parallel program so that the communication overhead is minimal.

The calculation of the communication overhead depends on the assumptions that are made. Measures of communication overhead have included the total distance the messages travel and the total distance adjusted by the message lengths. The best measure is the total execution time, however, even this is not defined uniformly. For example, should the total execution time be based on the minimum latency each message could encounter, or should it include the overhead of channel contention and possibly the misrouting of messages? Since the contention for a channel varies dynamically throughout the execution, it is difficult to determine the effects of contention on the total execution time. In addition, the amount of contention depends closely on the network topology and the routing algorithm. Even though it is hard to determine the amount of contention in a given mapping, contention can have a significant effect on the communication overhead and, hence, affects the total execution time. Thus, it is important to consider the channel contention when mapping the task graph.

1.6 Summary

In this chapter, we have given an overview of multiprocessor architectures and explained our motivation for addressing the problem of communication overhead in distributed-memory multiprocessors. Several techniques for addressing communication overhead have been presented, namely, the network topology, the switching technique, adaptive routing, and mapping. This dissertation focuses on reducing contention in distributed-memory multiprocessors. Results on both adaptive routing and mapping are presented. We also present a comprehensive approach that combines our research on adaptive routing with our research on mapping. Before presenting our research contributions, previous work in these two areas is described in Chapter II. An overview of the remainder of the dissertation is presented at the end of Chapter II.

CHAPTER II

Previous Work

In order to place the contributions of this dissertation in context, previous work on routing and mapping is reviewed. First, existing techniques for proving deadlock freedom are described. Our necessary and sufficient applies to any switching technique, so proof techniques for store-and-forward routing, virtual cut-through, and wormhole routing are described. Second, an overview of the existing routing algorithms for wormhole routing is described. For convenience, this presentation separates routing algorithms by topology. Finally, we describe many mapping heuristics. This description puts existing work in context as well as describing the niche filled by our work on mapping. At the end of this chapter, an outline of the remainder of the dissertation is presented.

2.1 Techniques for Proving Deadlock Freedom

For store-and-forward routing, most routing algorithms have used the proof technique proposed by Günther [45] to prove deadlock freedom. This proof technique requires an acyclic ordering of the buffers. Packets use the buffers in strictly increasing or strictly decreasing order, depending on how the ordering is defined. This acyclic ordering guarantees deadlock freedom by preventing a cycle in the buffer usage. Toueg and Steiglitz [98] have shown that this is necessary and sufficient for nonadaptive routing. Recently, Cypher and Gravano [12] have shown that an acyclic ordering of the buffers is *not* necessary for adaptive routing. They propose a simple routing algorithm that does not have an acyclic ordering of the buffers. The routing algorithm is deadlock-free, however, because it is impossible to generate a set of packets that can actually use the buffers in a cyclic manner. Dependencies among the packets themselves prevent the network from reaching this state. The authors call this type of cycle an unreachable configuration. Unreachable configurations cannot result in a deadlock configuration, because such configurations can never arise.

Pifarré *et al.* [82, 83] propose a proof technique for store-and-forward routing that permits cycles in the buffer ordering, provided that a subset of the routing algorithm has an acyclic buffer ordering and the routing algorithm is always able to choose a buffer from this subset. Duato [26] suggests that this proof technique is not valid, since a routing algorithm that is not deadlock-free can satisfy these requirements.

The proof techniques for store-and-forward routing can also be used for virtual cut-through. Every packet in a deadlock configuration is blocked. With virtual cut-through, a blocked packet is removed from a network and buffered, just like every packet is with store-and-forward routing. Thus, in any deadlock configuration, virtual cut-through reverts to store-and-forward routing.

Wormhole routing permits messages to span several routers in the interconnection network and, hence, simultaneously hold many channels, so the proof techniques developed for store-and-forward routing are not directly applicable to wormhole routing.

Designing deadlock-free routing algorithms for wormhole routing was simplified by Dally and Seitz [17] with a proof that an acyclic channel dependency graph guarantees deadlock freedom. Each vertex of the channel dependency graph is a virtual channel. There is a directed edge from one virtual channel to another if a message is permitted to use the second virtual channel immediately after the first. Deadlock freedom can be shown by assigning a numbering to the edges of the graph, ensuring that all channels are used in strictly increasing or strictly decreasing order.

Dally and Seitz proposed their proof technique for nonadaptive routing algorithms. Nonadaptive routing algorithms can be characterized by functions of the form $R : C \times N \times N \rightarrow C$, where the input channel, belonging to the set of channels C, and the current and destination nodes, belonging to the set of nodes N, define an output channel on which to route the message. An acyclic channel dependency graph has also been used as a basis for developing adaptive routing algorithms defined by relations of the form $R : C \times N \times N \rightarrow C^p$, where a set of output channels, rather than a single channel, is defined on which to route the message [5, 9, 13, 22, 37, 38, 39, 48, 66, 101].

Glass and Ni [39] and Boura and Das [5] have proposed techniques for generating deadlock-free algorithms. Both of these methodologies require an acyclic channel dependency graph. Glass and Ni proposed the *turn model* as a method of analyzing routing algorithms based on the permitted and prohibited dependencies from one channel to another. These dependencies are characterized as turns, with the set of possible turns defined by the topology. For example, meshes have 90° turns (when switching from a channel in one dimension to a channel in a different dimension), 0° turns (when switching from one channel to another channel in the same direction), and 180° turns (when switching from one channel to a channel in the opposite direction of the same dimension). The turn model groups the turns into cycles and breaks all cycles by prohibiting turns. This is equivalent to removing edges from the channel dependency graph. It is then necessary to show that cycles cannot be created from the remaining turns, based on the assumption that an acyclic channel dependency graph is required for deadlock freedom. Boura and Das propose a method of proving deadlock freedom by partitioning the channels into two sets and requiring messages to route completely in the first set before using channels in the second set. Although this technique is, in a sense, a more formal methodology than the turn model, it can generate only a subset of the routing algorithms that can be created using the turn model.

Duato [23, 25] proved that requiring an acyclic channel dependency graph is too restrictive for routing algorithms defined by relations of the form $R : N \times N \rightarrow C^p$, where the current node and the destination node, independent of the input channel, define the set of output channels on which to route a message. Cycles are permitted in the channel dependency graph if an acyclic *extended* channel dependency graph exists for some subset of channels that comprise a connected routing subfunction. An extended channel dependency graph contains both the direct and the indirect dependencies. Each edge in the channel dependency subgraph defines a *direct* dependency. An *indirect* dependency is a dependency between two channels in the subgraph that exists only because of the intermediate use of one or more channels not in the subgraph. We have used this methodology to propose a fully adaptive minimal routing algorithm for meshes. Berman, *et al.* [2] propose a fully adaptive torus routing algorithm with a routing relation of the form $R : C \times N \times N \rightarrow C^p$ that allows cyclic dependencies between the channels.

Su and Shin [95] propose a methodology for designing fully adaptive routing algorithms for a variety of topologies. Their approach requires an initial routing algorithm with an acyclic channel dependency graph and then adds an additional virtual channel to each physical channel. No restrictions are placed on the use of this extra channel. The routing algorithms proposed in the paper use dimension-order routing for the initial routing algorithm. Although these particular routing algorithms are deadlock-free, the proof technique does not, in general, produce deadlock-free routing algorithms, because indirect dependencies are ignored.

Dally and Aoki [16] prove deadlock freedom for a routing algorithm with cyclic dependencies by guaranteeing an acyclic *packet wait-for graph.* A packet wait-for graph is defined dynamically by the packets in the network and contains an edge from packet p_i to packet p_j if packet p_i is waiting for a channel occupied by packet p_j.

All these (valid) proof techniques provide only a sufficient condition for deadlock-free adaptive routing. Although Dally and Seitz [17] proved that an acyclic channel dependency graph is a necessary and sufficient condition for nonadaptive routing algorithms, determining what constitutes a necessary and sufficient condition for *adaptive* routing algorithms has remained an open problem.

Lin, McKinley, and Ni [64] propose a proof technique based on the observation that a routing algorithm is deadlock-free if none of the channels in the network can be held forever. If every message that uses a given channel is guaranteed to reach its destination, no matter which subsequent path (of those allowed by the routing algorithm) the message takes, then a deadlock configuration cannot arise from the use of this channel. Since sink channels cannot be part of a deadlock configuration, the proof starts with the sink channels and works backward through the network. If it is possible to show that no channel can be held forever by a message, regardless of the destination and path taken, then the routing algorithm is deadlock-free. This proof technique was proposed as a necessary and sufficient condition, although Duato [27] points out that only sufficiency is proved.

Duato [27] has recently proposed a necessary and sufficient condition for deadlock freedom for a restricted class of adaptive routing algorithms. This methodology applies only to routing relations of the form $R : N \times N \to C^p$. The necessary and sufficient condition requires the identification of a subset of channels, $C_1 \subseteq C$, which has an acyclic extended channel dependency graph. Unlike the sufficient condition, the set of channels in C_1 can differ for different source-destination pairs. Hence, Duato introduces the notion of cross dependencies. A cross dependency is a dependency from $c_i \in C_1$ to $c_j \in C_1$, where c_i and c_j are both in C_1, but for different source-destination pairs. As with the other dependencies, there are direct cross dependencies and indirect cross dependencies. A routing algorithm is deadlock-free if and only if some connected C_1 exists with no cycles arising from direct, indirect, direct cross, and indirect cross dependencies.

The proof technique proposed in this dissertation applies to routing relations of the form $R : C \times N \times N \rightarrow C^p$[1], while Duato's proof technique is more restrictive and applies only to routing relations of the form $R : N \times N \rightarrow C^p$. The latter routing relations can always be converted to routing relations of the former type by providing the same set of output channels for every input channel that the message could have used to reach that node (including input from the source when the source is the current node). In general, however, routing relations of the form $R : C \times N \times N \rightarrow C^p$ cannot be converted to routing relations of the form $R : N \times N \rightarrow C^p$, since the set of output channels can differ for the same destination when the message arrives on different input channels.

In addition to requiring the routing relation to be of the form $R : N \times N \rightarrow C^p$, Duato imposes two further restrictions on the routing algorithms for which his methodology provides a necessary and sufficient condition. Neither of these restrictions is imposed by our technique. First, the routing algorithm *must* provide a minimal path

[1]As discussed in Chapter IV, our methodology is actually more general than this.

between every pair of nodes, even for nonminimal routing algorithms. Second, the routing algorithm must be *coherent*. A routing algorithm is coherent if it permits every partial path from any source to any destination to be used by the same source to reach an intermediate node on the path or by an intermediate node on the path to reach the same destination. In other words, a coherent routing algorithm that allows a message from processor n_i to n_j to route through processor n_k must allow a message to use the partial path between n_i and n_j when routing from processor n_i to processor n_k or from processor n_k to processor n_j. Coherence may appear to be a modest restriction, however, simple adaptive routing algorithms exist that are not coherent. In addition, most nonminimal routing algorithms are not coherent. Whenever the routing algorithm permits a message to return to the source, or route through an intermediate node more than once, the routing algorithm is not coherent [28]. This is possible for any fully adaptive routing algorithm that permits nonminimal routing. Many partially adaptive nonminimal routing algorithms also permit this possibility.

Duato [26] has recently extended his necessary and sufficient condition to virtual cut-through and store-and-forward routing. This extension was done independently of the necessary and sufficient condition presented in Chapter IV. The requirements of a coherent routing algorithm and at least one minimal path between every source-destination pair have been removed, however, the proof technique still holds only for routing relations of the form $R : N \times N \rightarrow C^p$.

2.2 Routing Algorithms

Many routing algorithms have been proposed for interconnection networks. Most of the contributions on routing in this dissertation focus on wormhole routing. For this reason, routing algorithms for wormhole routing are reviewed in depth. Some discussion is also given to routing algorithms for store-and-forward routing. Our overview is limited to unicast routing algorithms, that is, routing algorithms for transmitting a message to a single destination. Multicast routing [65], which transmits a message to multiple destinations, is not reviewed.

2.2.1 Wormhole Routing on Meshes

Low-dimension meshes have relatively few channels and Dally [14] has shown that a few wide channels is better than many narrow channels. For this reason, the mesh is a popular topology in current distributed-memory multiprocessors, such as the Intel Paragon and the Symult 2010. Consequently, many adaptive routing algorithms have been developed for meshes. Although we discuss many mesh routing algorithms in detail, we first present a brief overview in Table 1 of the routing algorithms for meshes that were known when our result [90] was first published. In Table 1, VCs is used as an abbreviation for number of bidirectional virtual channels per router.

Glass and Ni [39, 41] used the turn model methodology to propose several partially adaptive routing algorithms for meshes. One example, *negative-first*, first routes a message adaptively in all the negative directions and then adaptively in all the positive directions. Another example is *west-first* for 2D meshes, which routes a message West

Table 1: Overview of Adaptive Routing Algorithms for Meshes

Author(s)	Fully Adaptive?	VCs for 2D Mesh	Comments
Chien & Kim [9]	Yes	6	Partially Adaptive for Higher Dimensions
Dally [13]	Yes	6	2D Mesh Only
Dally & Aoki [16]	Yes	k	2D Mesh with $k \times k$ nodes
Dally & Aoki [16]	Yes	8	'Dynamic' routing algorithm
Glass & Ni [38]	Yes	6	2D Mesh Only
Glass & Ni [39]	No	4	Roughly Half the Adaptiveness of Fully Adaptive
Jesshope, Miller, & Yantchev [48]	Yes	8	Number of Virtual Channels is Exponential in Dimension of Mesh
Linder & Harden [66]	Yes	6	

(if necessary) and then in any other direction without restriction. These routing algorithms also support nonminimal routing. Glass and Ni [40] show how to modify the negative-first routing algorithm to produce a routing algorithm that can tolerate at least $n-1$ faulty nodes on an n-dimensional mesh.

Boura and Das [5] use their methodology to propose a large class of partially adaptive routing algorithms. The methodology is fairly simple to apply, however, the only routing algorithms created using their methodology are symmetric to negative-first.

Many *fully* adaptive routing algorithms have been proposed for meshes. For the 2D mesh, the routing algorithms proposed by Chien and Kim [9], Dally [13], and Linder and Harden [66] all produce an equivalent fully adaptive minimal routing algorithm, *double-y*. Double-y requires two virtual channels in each Y direction (positive and negative) and one virtual channel in each X direction. Messages that need to route in the East direction use the East channel and the first set of channels in the Y dimension. Messages that need to route in the West direction use the West channel and the second set of channels in the Y dimension. This set of virtual channels has a total of sixteen 90° turns and four 0° turns. Double-y allows only eight of the sixteen 90° turns and prohibits all 0° turns.

Glass and Ni used the turn model to improve on double-y. They showed that double-y, although fully adaptive, imposes unneeded restrictions on the routing. They proposed the Maximally Fully Adaptive routing algorithm, *mad-y*, which makes better use of the virtual channels and hence improves adaptiveness. It was also shown that a fully adaptive routing algorithm with fewer virtual channels or fewer routing restrictions is not possible without allowing cycles in the channel dependency graph. Thus, it was argued that mad-y is the least restrictive fully adaptive routing algorithm for 2D meshes. Since an acyclic channel dependency graph is not required for deadlock freedom, this assertion does not hold. The routing algorithm for meshes we propose in this dissertation is more adaptive than mad-y.

Dally and Aoki [16] propose two adaptive routing algorithms for meshes. The first, *static*, labels each virtual channel with a dimension reversal number. Every time a message switches from a higher dimension to a lower dimension, the message moves to a virtual channel with a higher dimension reversal number. Once a message has reached a virtual channel with the highest dimension reversal number, the message uses dimension-order routing. Fully adaptive routing on a $k \times k$ mesh requires as many as k dimension reversals, so k virtual channels per physical channel are required for fully adaptive routing using the static algorithm. The second algorithm, *dynamic*, requires only two virtual channels per physical channel, one is used for nonminimal fully adaptive routing and the other for dimension-order routing. Once a message has used a virtual channel reserved for dimension-order routing, the message is no longer permitted to use the adaptive channels. With two virtual channels per physical channel, there are a total of thirty-two 90° turns and eight 0° turns. The dynamic routing algorithm allows only twenty of the thirty-two 90° turns and only four of the eight 0° turns.

Shen and Cheung [92] use the B-&-E (Basic and Extended) model to propose a fully adaptive routing algorithm for n-dimensional meshes. The virtual channels are partitioned into two sets, basic and extended. The basic channels are used in strictly increasing or strictly decreasing order, which requires an acyclic channel dependency graph. An extended channel can be used only if this ordering will not be violated between the previously used basic channel and the basic channel available to the message at the next router. They then prove that using dimension-order routing on the basic set of virtual channels allows the extended set of channels to be used without any restrictions.

Boura and Das [6] propose a fully adaptive routing algorithm for arbitrary dimension meshes that requires two virtual channels per physical channel. The routing algorithm uses the first set of virtual channels for dimension-order routing in the positive directions and then routes in the negative directions without any restrictions. No restrictions are placed on the second set of virtual channels. This routing algorithm is more adaptive than the routing algorithm proposed by Shen and Cheung [92].

Upadhyay *et al.* [99] propose a fully adaptive routing algorithm for 2D meshes. The routing algorithm combines the partially adaptive *negative-first* and *positive-first* routing algorithms proposed by Glass and Ni. Although the authors claim that their routing algorithm balances the channel usage across the network more evenly than other fully adaptive routing algorithms, twice as many channels are available when a message routes Northeast or Southwest than when a message routes Southeast or Northwest.

Liu and Chien [67] propose Hierarchical Adaptive Routing as an efficient way of supporting adaptive routing and fault tolerance in n-dimensional meshes. The virtual channels are partitioned into separate virtual networks. A message starts in the highest network and routes in that network until the message is delivered or the message is blocked. If the message is blocked, it is transferred to the next lower network and continues as before. Once a message reaches the lowest network, the message routes to the destination in that network. Since the networks are used in decreasing order, the routing algorithm is deadlock-free if the routing algorithm in the lowest virtual network is deadlock-free. The adaptiveness and fault tolerance can be increased by defining more virtual networks, however, this increases the number of virtual channels per physical channel. An advantage of Hierarchical Adaptive Routing is that the crossbars are relatively small, because each network is disjoint except for the connection between adjacent virtual networks. Smaller crossbars can result in a faster network cycle time.

Liu, Duato, and Thorelli [68] propose the *grouping technique* to provide a more balanced utilization of the channels in a 2D mesh. This approach starts with a fully adaptive routing algorithm that partitions the channels into separate virtual networks, such as *double-y*, and provides a *resource pool* of additional virtual channels. For example, with four virtual channels per physical channel, two of the virtual channels in the Y dimension form the resource pool. Both the East virtual network and the West virtual network can use these channels whenever they are available. This helps balance the channel utilization on the edges of the mesh, where the traffic is not evenly divided between the two virtual networks.

2.2.2 Wormhole Routing on Hypercubes

The hypercube topology has also been used in recent multiprocessors, such as the nCUBE-2 and nCUBE-3. For this reason, wormhole routing algorithms have also been proposed for hypercubes. In addition, any routing algorithm for meshes or tori can be used for hypercubes, because a hypercube can be considered a special case of either topology. Since torus and mesh routing algorithms are discussed elsewhere, only hypercube routing algorithms are reviewed in this section.

Partially adaptive routing algorithms for wormhole-routed hypercubes have been proposed by many authors [5, 9, 22, 37, 39, 43, 62, 63, 101]. Boura and Das [5] as well as Glass and Ni [37, 39] have proposed methods for producing partially adaptive routing algorithms for hypercubes. These routing algorithms are essentially restricted versions of their mesh routing algorithms. Only a single virtual channel per physical channel is required.

The partially adaptive routing algorithm for hypercubes proposed by Li [62] permits a message to route from one channel to another if the second channel is in a higher dimension or in the positive direction of a lower dimension. Thus, whenever a message needs to route h hops and requires the positive direction in the lowest dimension in which it needs to route, the message has a choice of h edge-disjoint paths. Otherwise, the message has a single edge-disjoint path. The routing algorithm randomly selects one of these paths and the message follows that path to the destination. In other words, the routing is *nonadaptive* once the message enters the network. Only one virtual channel per physical channel is required.

Gravano *et al.* [43] propose two partially adaptive hypercube routing algorithms. Only one virtual channel per physical channel is required by these routing algorithms. The first, *Non-Minimal*, routes messages along nonminimal paths. For each dimension, i, the message is first routed in a lower dimension, j, even if the message does not need to route in this dimension. The message is then routed in dimension i only if necessary. The choices for this lower dimension are predetermined. For example, when $i \geq 6$ then $j \in \{i-2, i-4, i-6\}$, when $4 \leq i < 6$ then $j \in \{i-2, i-4\}$, and when $i < 4$ no lower dimension is used. This routing algorithm is proposed as a method of breaking structured communication patterns by permitting a more uniform distribution of the network traffic. A minimal routing algorithm, *Subcubes*, is also proposed. The n-dimensional hypercube is partitioned into $n-k$ subcubes, where each subcube forms a k-dimensional hypercube. Messages are transmitted between subcubes by visiting nodes in the upward direction and then the downward direction, where the direction is defined relative to one of the subcubes. The message then routes within each subcube, using dimension-order routing or *positive-first*, for example.

A partially adaptive routing algorithm that supports both minimal unicast routing and multicast routing is proposed by Lin *et al.* [63]. The nodes of the hypercube are labeled such that a Hamiltonian path is created by following the nodes in sequential order. Messages are routed first in increasing label order and then decreasing label order.

The hypercube routing algorithm proposed by Draper and Ghosh [22] uses two virtual channels for each physical channel. Each message routes in dimension order along the first set of channels, but may skip some dimensions in which the message needs to route. The message then routes in dimension order along the second set of channels. The message can no longer skip dimensions and must wait for the channels to become free.

Yang and Tsai [101] also propose a hypercube routing algorithm that requires two virtual channels per physical channel. A message first uses any dimension in which it needs to route in a positive direction. When the message finishes with all such dimensions or finds them all busy, the message repeats this process with the negative directions. The message then switches to the second set of virtual channels and routes first in all remaining positive directions and then all negative directions, waiting for busy channels when necessary.

A fully adaptive hypercube routing algorithm has been proposed by several authors [23, 43, 64, 95]. This routing algorithm requires two virtual channels per physical channel. A message routes in dimension order along the first set of virtual channels. Each message also has the possibility of routing in any dimension that moves the message closer to the destination along the second set of virtual channels.

This routing algorithm permits only nonadaptive routing on the first set of virtual channels. In Chapter IV we show how to increase the adaptiveness by allowing partially adaptive routing on the first set of virtual channels.

2.2.3 Wormhole Routing on Tori

The torus is an increasingly popular choice for wormhole-routed multiprocessors. The Cray T3D, which uses a 3-dimensional torus topology, is one example. Deadlock-free routing on a torus is more complicated than on a mesh or hypercube. Because a torus has wrap-around channels, a deadlock configuration can be created in a single dimension. In fact, Glass and Ni [39] have pointed out that minimal routing on a torus with more than four nodes per dimension cannot be deadlock-free unless at least two virtual channels per physical channel are available. To solve this problem, Dally and Seitz [17] use dimension-order routing with two virtual channels per physical channel to prevent deadlock. Messages use the first virtual channel until a wrap-around channel is used. Messages then use the second virtual channel.

A fully adaptive minimal routing algorithm for 2-dimensional tori was proposed by Felperin *et al.* [31]. The routing algorithm requires eight virtual channels per physical channel. The network is divided into four classes: $X+Y+$, $X+Y-$, $X-Y+$, and $X-Y-$. Each message selects a class according to the directions in which it needs to route. For each class, there are two virtual channels per physical channel. The first virtual channel is used until the message uses the wrap-around channel and then the second virtual channel is used.

Berman *et al.* [2] propose a fully adaptive routing algorithm, **-Channels*, for an n-dimensional torus. The routing algorithm requires $10n-4$ unidirectional virtual channels per router. The first three unidirectional channels in each direction are used for dimension-order routing. With dimension-order routing on a torus, the second virtual channel is never used past the middle of the torus, because the routing is minimal and the second virtual channel is used only after the wrap-around channel has been used. These unused channels are removed to reduce the number of virtual channels. Two additional channels per direction (except for the lowest dimension) are used to support fully adaptive routing, so no restrictions are placed on the use of these virtual channels.

López and Duato [70] propose a routing algorithm for the 3D torus that is quite similar to the *-Channels routing algorithm. The only difference is that *multiple* virtual channels are available for adaptive routing. Simulation results are presented to show that with four additional virtual channels per physical channel, the network (an $8 \times 8 \times 8$ torus) can sustain much higher traffic rates before saturation and has a lower message latency.

Duato and López [29] propose a partially adaptive and a fully adaptive routing algorithm for tori. The partially adaptive routing algorithm routes messages in dimension order, but allows some messages to use either of the two virtual channels in that dimension. The fully adaptive routing algorithm adds an additional virtual channel per physical channel to the partially adaptive routing algorithm. No restrictions are placed on this additional channel.

Bopanna and Chalasani [4] propose two fully adaptive routing algorithms that are based on converting store-and-forward routing algorithms, proposed by Gopal [42], to wormhole routing. The first, *positive-hop*, starts each message in virtual channel zero and moves to virtual channel i on the i^{th} hop. The diameter of a k-ary n-cube is $n\lfloor k/2 \rfloor$, so $n\lfloor k/2 \rfloor + 1$ virtual channel per physical channel are required. The second algorithm, *negative-hop*, first labels each node with a one or two, so that no neighboring nodes have the same label. The message starts in virtual channel zero and moves to a higher numbered virtual channel every time the message routes from a node labeled two to a node labeled one. The maximum number of such moves on a k-ary n-cube is $\lceil n\lfloor k/2 \rfloor / 2 \rceil$, so $\lceil n\lfloor k/2 \rfloor / 2 \rceil + 1$ virtual channels per physical channel are required.

Rather than providing additional virtual channels to support deadlock-free routing on a torus, Kim, Liu, and Chien [54] use an abort/retry mechanism to terminate messages in a deadlock configuration. Since wormhole routing is used, messages are padded with extraneous data that follows the header through the network. Enough padding is provided so that the message header reaches the destination before all the padding enters the network. Once the padding has entered the network, the data flits can be transmitted. When deadlock occurs, a message is terminated, but before any data from the message has left the source.

2.2.4 Store-and-forward Routing

Nonadaptive store-and-forward routing uses the e-cube routing algorithm proposed by Sullivan and Bashkow [96], which uses the buffers in dimension order. Many adaptive routing algorithms for store-and-forward multiprocessors have been proposed. Only a few of the most recent or interesting of these routing algorithms are reviewed. The routing algorithms use multiple buffer classes for packets, and the buffer classes are used in acyclic order. Deadlock is prevented through a combination of restricting the use of buffer classes and using the buffer classes in acyclic order. Minimal routing with store-and-forward routing is defined by the number of nodes through which a packet routes, rather than the number of buffers. Hence, a routing algorithm that permits a packet to switch between buffer classes on the same node can still be minimal.

Gopal [42] proposed the first fully adaptive store-and-forward routing algorithms for arbitrary networks that require less than $H+1$ buffers, where H is the maximum number of hops in the network (for minimal routing, the network diameter). One example is the negative-hop algorithm previously discussed. This algorithm is extended to allow a buffer of class i to use any available buffer of class i or lower.

Felperin *et al.* [31] propose fully adaptive minimal routing algorithms for meshes and hypercubes. Two buffers, A and B, are required. A packet first routes in the A buffers, without restriction, as long as the packet needs to route in the *positive* direction of at least one dimension. The packet then moves to the B buffers and routes to the destination, in the negative directions, without restriction.

A fully adaptive minimal routing algorithm for tori was proposed by Cypher and Gravano [11]. Regardless of the number of processors or dimensions, only three buffers per router are required. This is the first store-and-forward routing algorithm for tori that requires a constant number of buffers. In addition, Cypher and Gravano [12] have proved that at least three buffers are required.

The routing algorithm initially places each packet in an A buffer and the packet remains in an A buffer as long as the packet needs to route in the positive direction along an internal channel (not a wrap-around channel). The packet can move to any A buffer (toward the destination), as long as this condition holds. The packet then routes in the B buffers as long as the packet needs to use a wrap-around channel in the positive direction or an internal channel in the negative direction. The packet then moves to the C buffers and routes in the C buffers until the destination is reached.

Konstantinidou and Snyder [56] propose *Chaos* routing to support adaptive non-minimal routing. Rather than using priority schemes to prevent livelock, livelock is avoided with high probability by misrouting a random packet when the buffers become full. Packets that have been waiting in a router input queue for a long period of time are moved to a central queue. A packet is misrouted only when this central queue becomes full. Since this occurs infrequently, few packets are misrouted.

2.3 Mapping Algorithms

Many mapping heuristics have been proposed for improving the performance of parallel processors. In this section, theoretical results, as well as mapping heuristics for general DAGs, are reviewed. Only compile-time mapping heuristics are considered.

The parallel algorithm 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 investigates the difficulties of assigning these tasks to the processors so that the parallel algorithm executes as quickly as possible.

Because message latency with wormhole routing is not sensitive to distance, it may seem that any mapping is equally good. Chittor and Enbody [10] consider the problem of mapping applications with a regular communication structure, such as a 2D mesh or hypercube, onto a 2D wormhole-routed mesh. The application model is not general, however, the authors are primarily interested in determining whether or not the mapping problem is important for wormhole-routed architectures. Channel contention is higher with wormhole routing than store-and-forward routing and the authors show that this leads to a substantial performance difference among mappings. Hence, the mapping problem is still important, even for wormhole-routed multiprocessors.

The problem of generating an optimal mapping is NP-hard when communication costs are considered. For this reason, a variety of mapping heuristics have been proposed. The seminal paper on the mapping problem is the mapping heuristic proposed by Bokhari [3]. This heuristic assumes that the number of tasks is less than or equal to the number of processors and that all the communication edges have the same weight. The goal of the heuristic is to minimize the total distance between the source and destination of all messages. Pairwise exchanges of tasks are used to reduce the total communication distance. Random reassignment of tasks is also used to avoid locally optimal mappings.

Lee and Aggarwal [60] also propose a mapping heuristic that applies only when the number of tasks matches the number of processors. A better model of communication is used. The communication edges can have different weights. In addition, the channel contention is considered, but only when communication occurs in distinct phases. The objective of the heuristic is to minimize the total overhead of all messages, thus all messages with the same weight have the same priority.

Both of these papers represent important initial work on the mapping problem, however, the solutions are not general. Restricting the number of tasks to match the number of processors is not realistic. In addition, reducing the communication cost does not necessarily reduce the total execution time.

Therefore, most recent research has focused on reducing the total execution time, rather than improving only the communication cost. Frequently, the transmission of messages can be overlapped with the execution of other tasks. Thus, the amount of communication overhead, which increases the execution time, is more important than the total communication cost. In addition, the impact on total execution time varies when messages are delayed. This is due to what is commonly referred to as the critical path. The critical path of a task graph is the longest path from a start node to a sink node. Any delay of messages on the critical path increases the total execution time of the task graph, however, messages that are not on the critical path can experience some delay before the total execution time increases.

It is well known that communication overhead is the main impediment to realizing satisfactory performance in most parallel applications. Thus, the model used for communication has a major impact on the quality of mappings produced by a mapping heuristic. In practice, a parallel architecture is not completely connected. For this reason, the overhead associated with communication between any two processors is not uniform. Additionally, channel contention can increase the transmission time of messages. This information can be used when mapping the algorithm to improve the execution time. Furthermore, if this data is ignored when mapping the task graph, the resulting schedule may be inefficient and the execution time estimate of the mapping may be incorrect.

Most recent mapping heuristics can be divided between list scheduling and clustering heuristics. List scheduling approaches are one-step techniques that extend results on list scheduling without communication costs to account for the communication overhead. Each task is assigned a priority level and the resulting list of tasks is scheduled in priority order. Communication overhead makes the priority dependent on the mapping, so the priority may be updated for unscheduled tasks as the mapping proceeds. Since list scheduling is a one-step approach, once a task is scheduled on a processor, it remains assigned to that processor. Tasks that are scheduled later are constrained by these earlier choices, since backtracking and modifying earlier task assignments is not permitted.

Clustering is a multi-step approach that first clusters the tasks on an unbounded number of completely connected processors. These clusters are then mapped to the processors, often after merging clusters until the number of clusters matches the number of processors. Each resulting cluster is then mapped onto a separate processor. The quality of the mapping can be tested and a different assignment of clusters to processors can be tried until an acceptable mapping is found.

Other approaches to mapping have been proposed. For example, graph decomposition methods can be used to attempt to restructure the task graph to match the underlying multiprocessor architecture. Khan, McCreary, and Jones [53] compare several list scheduling heuristics and clustering heuristics with a graph decomposition method of mapping proposed by one of the authors [74]. The comparison shows that existing list scheduling and clustering heuristics do not perform well for very fine-grain DAGs, where the communication cost is at least five times the computation cost. This suggests that either a different approach, such as graph decomposition, or better heuristics for clustering and list scheduling, are required for very fine-grain DAGs.

Both list scheduling and clustering approaches are examined in detail and mapping heuristics for both approaches are described. Before proceeding, however, we first examine some theoretical results on the mapping problem.

2.3.1 Theoretical Results

Finding an optimal mapping is, in general, NP-hard [32], even when there is an unbounded number of processors that form a completely connected architecture [79]. Papadimitriou and Yannakakis [79] propose a mapping heuristic that is within twice optimal when recomputation of tasks is permitted. The use of an unbounded number of completely connected processors may seem unreasonable, however, recall that this assumption is usually made as the initial step of the clustering approach to mapping. The clusters are later merged to match the number of processors, so a realistic mapping can be produced.

Papadimitriou and Yannakakis's mapping heuristic uses an unbounded number of processors. By applying the algorithm proposed by Thurimella and Yesha [97], the number of processors can be reduced to within a constant factor of optimal, while increasing the execution time to at most six times optimal. Mapping algorithms that generate optimal solutions for arbitrary DAGs, although requiring exponential time, have been proposed by Jung, Kirousis, and Spirakis [50], Markenscoff and Li [72], and Greenblatt and Linn [44]. The first two approaches produce an optimal schedule on an unbounded number of processors.

Various optimal mapping algorithms have been proposed for special classes of DAGs. For example, Kalpakis and Yesha [51] propose methods for optimally scheduling a bounded-degree forest on a mesh within a constant factor of optimal and a complete binary tree on a linear array within a constant of optimal. Löwe and Zimmermann [71] propose an algorithm for optimally scheduling any DAG, as long as all tasks satisfy the constraint that the computation time of every predecessor task is greater than the communication cost of every message from a predecessor.

2.3.2 List Scheduling Approaches

Kruatrachue and Lewis [58] propose a list scheduling heuristic that uses task duplication to reduce the execution time. The ratio of computation to communication is considered to derive the ideal grain size for a given multiprocessor. The tasks are first clustered into larger tasks until the granularity of the tasks reaches the grain size required for the target architecture. Communication overhead is considered when setting the task priorities, however, channel contention is ignored.

El-Rewini and Lewis [30] propose a list scheduling heuristic that considers communication overhead, including channel contention, when calculating the priority level of tasks. This heuristic is proposed for store-and-forward routing, although extensions to other switching techniques are possible. To measure contention, a table is kept that maintains the preferred shortest path between processors. This preference changes to reflect the use of channels in the network. The contention estimate is inaccurate for two reasons. First, the delay is overestimated. For example, a packet that takes five network cycles to route between adjacent processors and travels a distance of three hops is assumed to increase delay on this path by *fifteen* cycles. Second, the time when channels on the path are used is miscalculated. For example, the third channel in the above example has an immediate additional delay of five cycles, even though ten cycles pass before the message uses that channel. Hence, another message ready to enter the network could acquire this channel immediately and the first message could be blocked. Another disadvantage is that most interconnection networks do not allow the user to specify the message path, so the idea of selecting a single preferred path may not give a good estimate of contention.

Kon'ya and Satoh [57] propose a mapping heuristic for hypercubes that use non-adaptive circuit switching. List scheduling is used and the priority is determined by the latest starting time. The channel contention is included when determining the starting time of the task on a given processor. If the starting time is the same on several processors, the choice is based on which processor experiences either the least total distance or the least communication time for communicating results to this task.

Ghose and Mehdiratta [36, 75] propose a list scheduling heuristic that assigns priorities in a bottom-up manner rather than the top-down approach used in other list scheduling heuristics. The actual priority (and processor) of each task's ancestors is determined before scheduling a task, however, the processor choice of a task's predecessors is not known, so the advantage of a bottom-up approach is not apparent. The priority of the tasks should be recalculated to incorporate the communication overhead as tasks are scheduled, even with the bottom-up approach. The authors assume arbitrary routing on a hypercube using store-and-forward routing. Although the authors claim that their routing is deadlock-free, deadlock is actually quite likely, because there are no routing restrictions and messages can wait for busy channels.

Wang *et al.* [100] propose a list scheduling heuristic that considers channel contention when selecting the best processor for each task. Store-and-forward routing is assumed, although the result can be extended to other switching techniques. By considering the communication cost, a better processor can be chosen, however, the task priorities are assigned independent of the communication cost and the tasks are scheduled on processors that may not be idle. Since the communication cost does not change the priority, critical tasks could get a lower priority than non-critical tasks. This is possible because the set of critical tasks depends on the communication cost.

Sih and Lee [93] propose a list scheduling heuristic that incorporates a better model of communication. The authors divide their heuristic into two parts, mapping the computation and scheduling the communication. These two aspects are considered concurrently in order to generate a better mapping. A dynamic priority level is assigned to each task. The dynamic level is adjusted according to the availability of processors and network channels. The task with the highest priority chooses the best processor, whether or not the processor is currently available. The communication model includes contention, but the overhead of sending messages across multiple channels is ignored. This is only slightly inaccurate for wormhole routing and circuit switching, but highly inaccurate for store-and-forward routing. Another drawback is the use of list scheduling. Since backtracking is not permitted, finding good paths for critical messages may become difficult. The channels may already be in use by messages that are not critical. The authors have extended the heuristic to support heterogeneous multiprocessors [94].

2.3.3 Clustering Approaches

Sarkar and Hennessy [86] proposed the first clustering heuristic, which they called "internalization pre-pass". Their clustering heuristic first sorts the edges in decreasing order of communication cost. The edges are examined in order and the source and destination tasks are clustered if the total execution time does not increase. The complexity of this clustering heuristic is $O(E(V+E))$.

Kim and Browne [55] propose a mapping heuristic that constructs linear clusters and then merges clusters that either have no tasks with overlapping execution times or the communication cost between clusters is greater than the resulting increase in execution time. The clusters are then sequentially mapped onto the processors, using pairwise exchange and other techniques. The objective is to maintain the structure of the original clustered graph. Hence, the heuristic attempts to assign communicating clusters on neighboring processors.

Yang and Gerasoulis [103, 105] have proposed a clustering heuristic, called the Dominant Sequence Clustering (DSC) algorithm, which has a complexity of only $O((V+E)\log V)$. They prove that linear clustering is better for coarse-grain DAGs. DSC is known to produce optimal schedules for fork DAGs, join DAGs, and coarse-grain tree DAGs. The authors show that for coarse-grain DAGs, the clustering produced by DSC is within twice the optimal clustering. In reality, mapping each task to a separate processor produces a schedule within twice optimal, as does any clustering heuristic that is guaranteed not to increase the execution time after any clustering step. The low time complexity and the optimal results for some DAGs make DSC an attractive clustering heuristic.

The authors have incorporated DSC into their PYRROS [104] mapping system, which converts a task graph into executable code for a distributed-memory multiprocessor. The first step of the mapping algorithm uses DSC to create the clusters. The clusters are then merged, using the work profiling method of George, Heath, and Liu [35], until the number of clusters matches the number of processors. In essence, the clusters are merged so that each cluster has approximately the same amount of computation. These clusters are then mapped onto the processors using Bokhari's mapping heuristic. The tasks on each processor are then assigned a priority level, based on their distance from the sink nodes of the DAG. The distance between processors is used when determining the transmission time of messages and this overhead is included in the priority calculation.

Darbha and Agrawal [19] propose a clustering heuristic that uses task duplication and can generate optimal schedules for some coarse-grain DAGs. Only the initial clustering step is performed, the merging of clusters and the mapping onto a real multiprocessor are not considered. Task and edge duplication is used to convert the task graph into a tree DAG, which is then used for the scheduling. The complexity is claimed to be $O(V + E)$, however, in the worst case, tasks and edges in the original graph are replicated an exponential number of times. Hence, the claim of linear time complexity does not hold. In [18], the authors extend the heuristic to allow a task to have different execution times, corresponding to a branch or loop that may not be predictable during compilation. Allowing alternative execution times increases the complexity of scheduling, so the authors allow only one task to have a variable execution time in their experiments.

Ahmad and Kwok [1] propose a clustering heuristic that allows task duplication. Task duplication is used for ancestors of tasks on the critical path. The duplication is done in decreasing order of message arrival time, so that the starting time of the critical task can be reduced as much as possible. Ancestors of a task are duplicated on this processor as long as the starting time of the task does not increase. The authors propose another clustering heuristic in [59]. The tasks in a cluster are given a range of starting times, but the starting time of a task is not fixed until all tasks are scheduled. This allows tasks with a higher priority to start earlier by starting less critical tasks later. The tasks in a cluster are given a total order by introducing dummy edges between tasks. This ensures the priority is maintained. The authors show that fewer clusters are created using their heuristic than, for example, the DSC heuristic. The total execution time of the task graph is also better than other approaches. It is not clear, however, that reducing the number of clusters is a good idea, since this restricts the flexibility of merging clusters when mapping to the real multiprocessor. The time complexity of the first heuristic is $O(pEV)$ and the time complexity of the second heuristic is $O(pV(E+V))$, where p is the number of clusters. Because $p \in O(V)$, the time complexity is $O(V^2E)$ and $O(V^2(E+V))$, respectively.

Yang, Bic, and Nicolau [102] propose a heuristic for mapping clustered task graphs onto processors. Store-and-forward routing is assumed. The clusters are mapped onto the processors using an iterative approach. Initially, the clusters with critical edges are assigned so that as many critical edges as possible are between neighboring processors. The clusters without critical edges are assigned, if possible, to a processor with a cluster that communicates with this cluster. Each cluster without critical edges is then randomly reassigned until an optimal mapping is achieved or p (the number of processors) mappings have been tried. An optimal mapping is defined as the execution time on a completely connected architecture with p processors. There are at least three drawbacks to this approach: (1) For large DAGs, most clusters have at least one critical edge, so it is possible that only a single mapping is tried. Even if multiple mappings are tried, it is likely that the majority of the clusters have critical edges. Hence, most clusters are never moved. (2) The set of critical edges depends on the mapping, because the communication cost depends, in part, on the distance between processors. The dynamic nature of the critical edge set is ignored. (3) The heuristic ignores channel contention, so the estimate of execution time is inaccurate and the chosen mapping may not be the best of the generated mappings.

2.3.4 Explicitly Parallel Programs

Explicitly parallel programs already contain communication primitives, such as *send*() and *receive*(), and have been designed specifically for parallel computation. This is slightly different from the DAG model previously considered. With the DAG model, the task graph contains implicit parallelism that must be extracted for efficient parallel computation. An explicitly parallel program must still be mapped to reduce the communication overhead, but issues such as data distribution, which are typically ignored with the DAG model, are addressed by the task structure defined when the parallel program is written. We briefly review some results on mapping explicitly parallel programs.

Chaudhary and Aggarwal [7] propose a mapping heuristic for scheduling explicitly parallel programs. The heuristic considers contention and is applicable to any topology. Store-and-forward routing is assumed. The authors introduce the notion of feasible mapping and argue that a mapping is feasible only if the communication generated by the mapping does not lead to a deadlock in the network and every processor has enough buffer space to hold the messages it receives. Graph theory, mathematical programming, and heuristic methods are combined to generate a mapping.

In a series of papers [20, 21, 78], Dixit-Radiya and Panda present a comprehensive approach to the problem of mapping explicitly parallel programs. They propose a mapping heuristic, based on clustering, for explicitly parallel programs in [21]. Rather than using the DAG model, the Temporal Communication Graph model proposed by Lo *et al.* [69] is used. This model facilitates the overlapping of communication and computation by keeping track of when messages occur. Unlike the DAG model, tasks can send and receive messages at any time, rather than receiving messages only before execution and sending messages only after execution. This flexibility allows the task structure to implicitly maintain regular data distributions. Given a set of ready tasks on a processor, the task with the minimum latest starting time is executed.

Three clustering heuristics are proposed. Each iteratively merges a pair of clusters until the number of clusters matches the number of processors. The first heuristic chooses the pair of clusters that maximizes the reduction in total execution time of the parallel program. The other two heuristics reduce the time complexity of clustering by estimating the improvement in execution time without recomputing the change in the overall schedule length and by not recomputing the critical path and occurrence time of messages after each clustering step. Simulation results show that all three heuristics perform better than the heuristic proposed by Chaudhary and Aggarwal [7].

The authors [20] also consider the problem of mapping these clusters onto the processors. Most current multiprocessors use wormhole routing, and channel contention is the primary cause of additional message latency in wormhole-routed multiprocessors. Thus, an initial mapping is made based on minimizing the distance between clusters that share the most data. The execution is then simulated, measuring the channel contention. Clusters with messages on the critical path that experience contention are reassigned to neighboring processors. The mapping is simulated after each pairwise exchange. If mapping to neighboring processors does not result in a satisfactory mapping, then processors within d hops ($d > 1$) are tried. Results are presented for a variety of routing algorithms (nonadaptive, partially adaptive, fully adaptive) on meshes and hypercubes. Task graphs with varying amounts of parallelism and differing ratios of computation to communication are tested. For most graphs, the initial mapping is relatively good, although the iterative refinement improves the mappings.

The authors [78] also address the issue of message ordering when multiple messages are waiting to be injected. The messages are injected in decreasing order of latest starting time. Compared to sequential message ordering, this ordering results in a substantial improvement in the execution time.

2.3.5 Summary

Although many researchers have recognized that channel contention can be a major factor in communication overhead, no research has addressed this problem for clustered task graphs. List scheduling approaches have the disadvantages of the dynamic changing of priorities and the inability to adjust the processor assignment of previously scheduled tasks. The priority levels can be dynamically adjusted, but this increases the time complexity of the mapping heuristic.

Once a task graph has been clustered, the clusters can be treated as a whole. This limits the mapping flexibility to some extent, but also reduces the time complexity of mapping. This reduction in complexity makes it feasible to try multiple mappings of the same task graph. In Chapter V, we propose a heuristic for clustered task graphs that considers channel contention when selecting a mapping.

Some approaches to reducing contention have relied on selecting fixed paths for messages. The issue of selecting fixed paths has been explicitly considered by Lee and Kim [61]. They propose a path selection scheme for routing messages through the network. The problem of selecting paths to minimize contention is NP-hard, so the authors consider a limited set of paths (four or eight) for a message and choose the one with the least contention. Although the authors show their approach works well compared to other path selection schemes, the performance relative to fully adaptive routing is not clear. Furthermore, specifying the routing path is not possible in most current multiprocessors, although this capability is sometimes available in real-time systems.

2.4 Dissertation Overview

In this dissertation, our primary focus has been on adaptive routing, with a secondary emphasis on mapping. We address several fundamental open problems related to adaptive routing and proofs of deadlock freedom. Our main focus is on routing algorithms for wormhole-routed multiprocessors. Focusing on wormhole routing has two important advantages. First, wormhole routing has become the switching technique of choice in modern distributed-memory multiprocessors such as the Intel Paragon, the Cray T3D, the MIT J-machine, the Caltech MOSAIC, and the nCUBE-2/3. Second, the problems encountered with wormhole routing are often more difficult than the problems that arise with other switching techniques. Hence, the results obtained for wormhole routing can often be generalized to other switching techniques. This has allowed us to provide more comprehensive results, even while focusing on issues that are of current interest.

A significant portion of this dissertation is devoted to addressing the issue of deadlock-free routing. When designing routing algorithms, a few questions immediately arise: What are the least restrictive properties a routing algorithm must have to guarantee deadlock freedom? In other words, what is a necessary and sufficient condition for deadlock-free routing? For a given network topology, how restrictive must a routing algorithm be to guarantee deadlock freedom? Similarly, for a given network topology, what is the minimum number of virtual channels required to support a given degree of adaptiveness? Our research on adaptive routing algorithms focuses on these questions.

In Chapter III, an optimal fully adaptive routing algorithm for n-dimensional meshes is proposed. This routing algorithm is optimal in both the number of virtual channels required and the number of restrictions placed on those virtual channels. Furthermore, we prove that any fully adaptive routing algorithm that is optimal in both of these ways must be a symmetric case of the routing algorithm that we propose. In other words, the proposed routing algorithm is uniquely optimal.

In Chapter IV, a necessary and sufficient condition for proving deadlock freedom is proposed. This proof technique is universal, in the sense that it is applicable to any switching technique that uses routing restrictions to guarantee deadlock freedom. To demonstrate the practical advantages of our methodology, a routing algorithm for wormhole-routed hypercubes is proposed that is substantially more adaptive than any previously known hypercube routing algorithm. In addition, a nonminimal partially adaptive routing algorithm for meshes is proposed. To demonstrate the universality of our technique, we propose a fully adaptive routing algorithm for n-dimensional packet-switched meshes that is more adaptive than the routing algorithm proposed by Pifarré *et al.* [83].

In Chapter V, we present a contention-sensitive mapping heuristic for the compile-time mapping of clustered task graphs. The heuristic uses simulation to determine the channel contention and uses this information to modify the current mapping, as well as select the best mapping. Simulation results for packet-switched hypercubes are presented to compare our heuristic with an existing mapping heuristic [102]. These simulation results support our claims that channel contention can have a significant effect on the execution time of a mapping and ignoring channel contention leads in an inaccurate estimate of the execution time of a mapping.

In Chapter VI, the work in the previous three chapters is combined by studying the performance of the mapping heuristic with the routing algorithms we have proposed. The routing algorithms we have proposed are compared with other routing algorithms for the mappings that are generated. In order to provide a more realistic comparison, DAGs derived from the Cholesky factorization of sparse matrices are used.

Finally, Chapter VII briefly summarizes the work contained in this dissertation. Future extensions to this work are also described. The discussion concludes with an evaluation of the applicability of this work, given the constant changes in technology.

CHAPTER III

Optimal Fully Adaptive Minimal Wormhole Routing for Meshes

Recent research on adaptive routing algorithms has focused on reducing the number of virtual channels required to support adaptive routing and on improving the use of these virtual channels by reducing the restrictions placed on the use of the virtual channels. In this chapter, theoretical results for fully adaptive routing on meshes are presented which demonstrate tight lower bounds on the number of virtual channels and the number of restrictions on the virtual channels. We prove these claims for fully adaptive minimal routing on meshes with uniform routers. Besides providing a theoretical framework for addressing these two issues, the routing algorithm proposed in this paper improves on previous routing algorithms both in the number of virtual channels required and in the number of restrictions placed on the use of the virtual channels. In other words, the optimal routing algorithm requires fewer virtual channels, and is more adaptive, than *any* other routing algorithm. We also prove that, except for symmetric routing algorithms, the routing algorithm we propose is the *only* routing algorithm that is optimal in both these ways. The results presented in this chapter have been published in [90].

We first describe our notation and the set of standard assumptions that are made about wormhole routing. Our routing algorithm is presented for the 2D mesh in Section 3.3 with the proofs of optimality presented in Section 3.4. A detailed comparison of our routing algorithm and mad-y, which was previously the most adaptive routing algorithm for 2D meshes is presented in Section 3.5. We then generalize our result to n-dimensional meshes in Section 3.6 and compare our routing algorithm with other routing algorithms for n-dimensional meshes. Finally, we show how any extra virtual channels can be incorporated into our algorithm, while maintaining optimality.

3.1 Notation

The following conventions are used throughout the chapter. Channels are assumed to be bidirectional. All channels, whether physical or virtual, are referred to as virtual channels. N, S, E, and W are used to indicate the appropriate direction, with, for example, N–W used to indicate that a message has switched from the North direction to the West direction. The symbol VC_{nd} denotes virtual channel n in the d direction. For example, $VC_{1\mathrm{N}}$ is virtual channel one in the North direction. The channel number is omitted when there is a single virtual channel in that direction.

3.2 Assumptions and Definitions

Several assumptions and definitions are introduced to facilitate the presentation of the routing algorithms. These are standard assumptions made when proving deadlock freedom for wormhole routing algorithms and have also appeared in [17, 25]. The definitions are also used in Chapter IV.

1. A node can generate messages of arbitrary length destined for any other node at any rate.

2. A message arriving at its destination is eventually consumed.

3. Since wormhole routing is used, once a channel queue accepts the header flit of a message, it must accept all the flits of the message before accepting any flits from another message.

4. A channel queue cannot contain flits belonging to more than one message at a time. The channel must transmit the tail flit of the current message before the channel queue accepts the header flit of the next message.

5. A router arbitrates among messages simultaneously requesting the same output channel, however, messages already waiting for a channel are chosen in an order that prevents starvation.

Definition 3.1 An *interconnection network* I is a strongly connected directed multigraph, $I = G(N, C)$, where the vertices, $n_i \in N$, are the processors and the arcs, $c_i \in C$, are channels that connect neighboring processors. Each channel, c_i, can transmit messages from one processor, denoted s_i, to a neighboring processor, denoted d_i.

Definition 3.2 A *routing relation* has the form $R : C \times N \times N \rightarrow C^p$ and specifies a set of output channels based on the input channel, the current node, and the destination of the message.

Definition 3.3 A *selection function* has the form $S : C \times C^p \times F^p \rightarrow C$ and chooses a single output channel from a set of output channels based on the input channel and the status of the output channels. F represents the possible states of an output channel. For fault-tolerant routing, $F = \{\text{free, busy, faulty}\}$; otherwise, $F = \{\text{free, busy}\}$.

Definition 3.4 A *routing algorithm* R_A on interconnection network I is represented by $R_A(n_i, n_j)$ and for each source-destination pair defines the set of possible paths taken by a message. The routing is accomplished by repeated application of a routing relation and then a selection function at each router between the source and destination of the message. The routing algorithm may be adaptive or nonadaptive; minimal or nonminimal; fault-tolerant or not fault-tolerant.

3.3 Optimal Minimal Routing

Requiring an acyclic channel dependency graph places unnecessary restrictions on the routing algorithm. Mad-y, which was previously the most adaptive routing algorithm for meshes, requires an acyclic channel dependency graph. Thus, mad-y is not the least restrictive routing algorithm for 2D meshes. A new routing algorithm, the Optimal Fully Adaptive routing algorithm, which has substantially fewer restrictions, is proposed. Since there are cycles in the channel dependency graph, the turn model cannot be used to prove deadlock freedom. An extended version of the turn model notation is used, however, to depict the routing restrictions.

A block diagram of the router with two virtual channels in the North and South directions is shown in Figure 8(a). The virtual channels in the North and South directions are differentiated by marking $VC_{1\mathrm{N}}$ and $VC_{1\mathrm{S}}$ with a single dash and $VC_{2\mathrm{N}}$ and $VC_{2\mathrm{S}}$ with a double dash. This set of virtual channels has sixteen 90° turns and four 0° turns. *Unrestricted* turns are indicated by solid lines, *restricted* turns by dashed lines, and *prohibited* turns by dotted lines. The term *permitted* turns is used for the combination of restricted and unrestricted turns. The constraints imposed by the Optimal Fully Adaptive routing algorithm, called *opt-y*, can be summarized succinctly: a message that needs to route further in the West direction cannot use $VC_{1\mathrm{N}}$ or $VC_{1\mathrm{S}}$. Opt-y has the following two sets of constraints (See Figures 8(b) and 8(c)):

- Two 90° turns, N–W using $VC_{1\mathrm{N}}$ and S–W using $VC_{1\mathrm{S}}$, are prohibited.
- The 0° turns from $VC_{2\mathrm{N}}$ to $VC_{1\mathrm{N}}$ and from $VC_{2\mathrm{S}}$ to $VC_{1\mathrm{S}}$ are allowed only when the message does not need to route further West.

The following restrictions arise solely from the previous constraints:

- The 90° turns W–S using $VC_{1\mathrm{S}}$ and W–N using $VC_{1\mathrm{N}}$ are allowed only when the message does not need to route further West. These turns are restricted only because the N–W turn from $VC_{1\mathrm{N}}$ and S–W turn from $VC_{1\mathrm{S}}$ are prohibited.
- The 0° turns from $VC_{1\mathrm{N}}$ to $VC_{2\mathrm{N}}$ and from $VC_{1\mathrm{S}}$ to $VC_{2\mathrm{S}}$ are allowed only when the message does not need to route further West, because $VC_{1\mathrm{N}}$ and $VC_{1\mathrm{S}}$ are not used until the message has completed routing West.

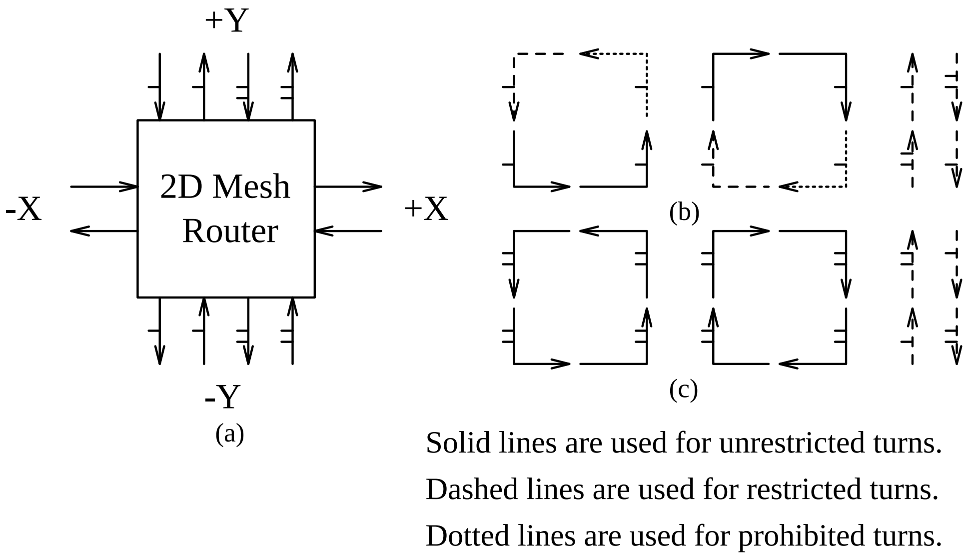


Figure 8: Optimal Fully Adaptive Routing

- $VC_{1\mathrm{N}}$ and $VC_{1\mathrm{S}}$ cannot be used by the router at the source if the message needs to route West.

Several important properties are proved for opt-y:

- Opt-y is fully adaptive.
- Opt-y is deadlock-free.
- Opt-y requires only the minimum number of virtual channels.
- Opt-y imposes only the minimum number of routing restrictions on these virtual channels.
- Opt-y is the only algorithm to satisfy all of the previous properties (except for algorithms symmetric to opt-y).

Theorem 3.1 *Opt-y is fully adaptive.*

Proof. With minimal routing, a message always routes closer to the destination. When the source and destination differ in only one dimension, the message can use the virtual channel(s) in the appropriate direction; the single virtual channel in the East and West directions or either virtual channel in the North and South directions. For a destination that is Northeast (Southeast) of the source, the message can route adaptively along the North (South) and East virtual channels. For a destination that is Northwest (Southwest) of the source, the message can route adaptively along $VC_{2\mathrm{N}}$ ($VC_{2\mathrm{S}}$) and the West virtual channel. Once the message has completed routing West, it can use either of the North (South) virtual channels and switch between them. $\square$

A message can use $VC_{1\mathrm{N}}$ or $VC_{1\mathrm{S}}$ only when the destination is not West of the current node. Otherwise, it must choose one of the other virtual channels. This restriction is independent of the input channel, so opt-y has a routing relation of the form $R : N \times N \to C^p$. Therefore, Duato's result [23] can be used to prove that opt-y is deadlock-free.

Using Duato's terminology [23], the routing algorithm is denoted R and the set of channels used by R is denoted C. There is some subset of channels, $C_1 \subseteq C$, that defines a routing algorithm $R_1 \subseteq R$, *i.e.*, R_1 is the routing algorithm R restricted to using the set of channels C_1. When there are cycles in the channel dependency graph, it is impossible to provide a numbering of the channels that guarantees the channels are used in always increasing or always decreasing order. Instead, Duato has shown [23] that any routing algorithm of the form $R : N \times N \rightarrow C^p$ is deadlock-free if it satisfies the following three conditions:

- Routing algorithm R_1 is connected.
- Routing algorithm R_1 is acyclic and therefore deadlock-free.
- The additional channels $(C - C_1)$ of routing algorithm R do not introduce cycles in the extended channel dependency graph of C_1.

The idea behind the proof is that cycles are permitted in the channel dependency graph if every message always has the possibility of switching to an acyclic path. The set of output channels, C^p, always includes at least one channel in C_1, so a deadlock-free path is always available. Whenever all channels in C^p are busy, the output selection policy either defers selecting a channel or chooses a busy channel in $C_1 \cap C^p$[1]. For opt-y, C_1 consists of all the virtual channels except $VC_{2\mathrm{N}}$ and $VC_{2\mathrm{S}}$. Consequently, R_1 uses all the permitted 90° turns shown in Figure 8(b). R_1 is *west-first* (with minimal routing) proposed by Glass and Ni [39].

Lemma 3.1 *Routing algorithm R_1 is connected.*

Proof. The only restriction imposed by R_1 (other than minimal routing) is that a message cannot use a West channel after using a North or South channel. Therefore, XY routing is a subset of R_1 and the lemma follows immediately from the fact that XY routing is connected. □

[1]This requirement can be relaxed by using the necessary and sufficient condition in Chapter IV.

Lemma 3.2 *Routing algorithm R_1 is acyclic and therefore deadlock-free.*

Proof. Messages must route West before routing North or South. Since there are no dependencies from a channel in the Y dimension to a channel in the West direction, R_1 is acyclic. □

Lemma 3.3 *The additional channels of routing algorithm R do not introduce cycles in the extended channel dependency graph of C_1.*

Proof. The only possibility for a cycle in C_1 is for an *indirect* dependency to be introduced by using $VC_{2\mathrm{N}}$ or $VC_{2\mathrm{S}}$. First notice that R_1 is fully adaptive for all messages except messages with a destination West of the source. Indirect dependencies that allow a use of the channels different from R_1 could arise in only two ways: (1) A message uses VC_{W} after using $VC_{1\mathrm{N}}$ or $VC_{1\mathrm{S}}$. This cannot occur, since the routing algorithm, R, prohibits the use of $VC_{1\mathrm{N}}$ or $VC_{1\mathrm{S}}$ by any message that needs to route further West. (2) A message uses some VC_{W} and later uses another VC_{W} in a different row of the mesh. This is possible only because a message that is routed West can use $VC_{2\mathrm{N}}$ or $VC_{2\mathrm{S}}$ and later route West again, however, this indirect dependency does not cause a deadlock. The channel dependency graph for C_1 has no dependencies from a channel in the East, North, or South directions to a channel in the West direction, so the West channels are always used before any other channel in C_1. Hence, these indirect dependencies create new dependencies among only the West virtual channels. Since a mesh has no wrap-around channels and minimal routing is used, there are no cycles using only the West virtual channels. Therefore, the extended channel dependency graph of C_1 is acyclic. □

Theorem 3.2 *Opt-y is deadlock-free.*

Proof. The proof follows immediately from lemmas 3.1 – 3.3. □

3.4 Proofs of Optimality

Opt-y is optimal for fully adaptive minimal routing in two ways: (1) It requires only the minimum number of virtual channels per router. (2) It imposes only the minimum number of routing restrictions on the virtual channels. Two reasonable and standard assumptions are made in connection with proving the optimality of opt-y. First, optimality holds for the number of virtual channels at only the interior nodes of the mesh. Nodes on the periphery of the mesh need to be connected to other nodes in only two or three directions, so fewer virtual channels could be used for these routers. Second, all the nodes use identical routers and thus the same routing algorithm.

The optimality proofs are greatly simplified by proving that *all* configurations with fewer than six virtual channels and *most* configurations with six virtual channels are either not deadlock-free or not fully adaptive. (This was proved by Glass and Ni [38] only for routing algorithms with an acyclic channel dependency graph.)

Note: The following conventions are used for figures showing deadlock configurations. In each figure, the routers are represented as circles and the channels as edges connecting the routers. The source and destination of message i are labeled Si and Di, respectively. Virtual channels acquired by message i are labeled Mi. Bidirectional channels are assumed, so arrowheads are used to indicate the direction of each message. Messages on the left side of a vertical channel use VC_1 and messages on the right side use VC_2. Similarly, messages above a horizontal channel use VC_1 and messages below use VC_2.

Theorem 3.3 *Deadlock-free fully adaptive wormhole routing on a 2D mesh requires a second virtual channel in two opposite directions, either North and South or East and West.*

Proof. Consider the potential deadlock configuration shown in Figure 9(a). All four messages in the cycle are waiting for a virtual channel in the North or West direction. If there is only a single virtual channel in the North and West directions, then the routing algorithm is not deadlock-free. This applies to *any* fully adaptive minimal routing algorithm. Similarly, Figure 9(b) shows a deadlock configuration if there is a single virtual channel in the North and East directions. Figure 9(c) shows a deadlock configuration if there is a single virtual channel in the South and West directions. Figure 9(d) shows a deadlock configuration if there is a single virtual channel in the South and East directions. Two additional virtual channels are required to resolve all four deadlock cases and the two virtual channels must be added in opposite directions. For example, the first two deadlock configurations can be avoided by adding a virtual channel in the North direction and the last two deadlock configurations can be avoided by adding a virtual channel in the South direction. □

By theorem 3.3, opt-y requires only the minimum number of virtual channels for deadlock-free fully adaptive routing. The remainder of this section is used to show that opt-y has the fewest restrictions of *any* fully adaptive routing algorithm with

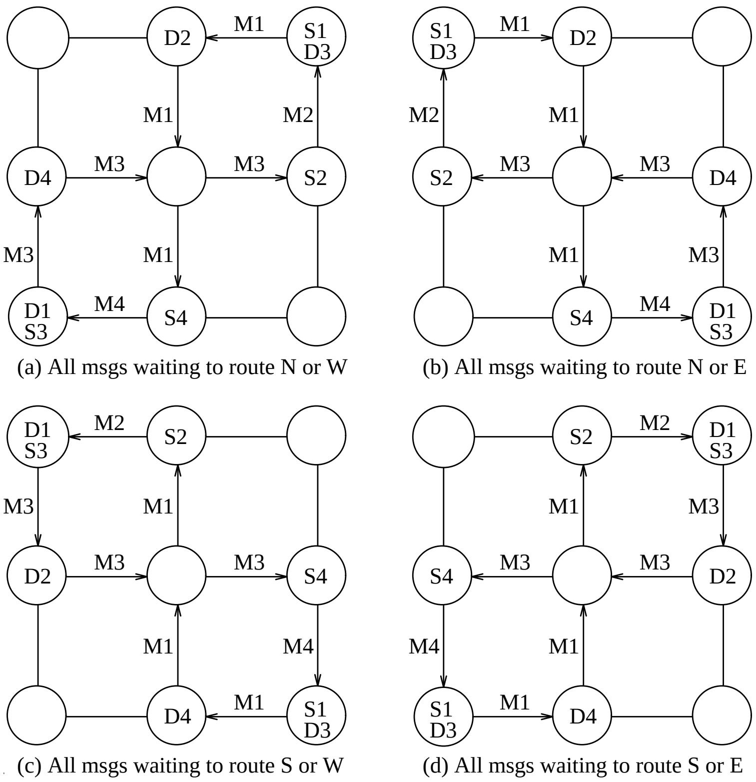


Figure 9: Potential Deadlock Configurations

six virtual channels, and that, ignoring symmetry, *all* other fully adaptive routing algorithms have more restrictions than opt-y.

For adaptive routing, a cycle in the channel dependency graph is not a sufficient condition for deadlock. A *knot*, however, is a sufficient condition for deadlock, so the proofs of deadlock are based on the existence of knots (rather than cycles) in the channel dependency graph. A knot exists when a set of messages forms a cycle and no message in the set can use a channel outside the set. In other words, a knot is a cycle with the additional requirement that *all* messages in the knot have paths to only channels held by other messages in the knot.

Lemma 3.4 *At least two* $90°$ *turns must be prohibited to make the routing algorithm deadlock-free.*

Proof. Knots can occur when a set of messages routes in the clockwise or counter-clockwise directions, so it is necessary to prevent both of these knots. At least one $90°$ turn must be prohibited to prevent each knot, so at least two $90°$ turns must be prohibited. □

Since, by lemma 3.4, at least two $90°$ turns must be prohibited and opt-y prohibits only two, the only other routing algorithms that need to be considered are those that prohibit just two $90°$ turns. Ignoring symmetry, there are only three such routing algorithms: *north-last*, *negative-first*, and *west-first* [39]. In addition, by theorem 3.3, only configurations with a second virtual channel in either the North and South or the East and West directions need to be considered. Therefore, there are six different routing algorithms to consider. Opt-y, which uses *west-first*, is deadlock-free. A

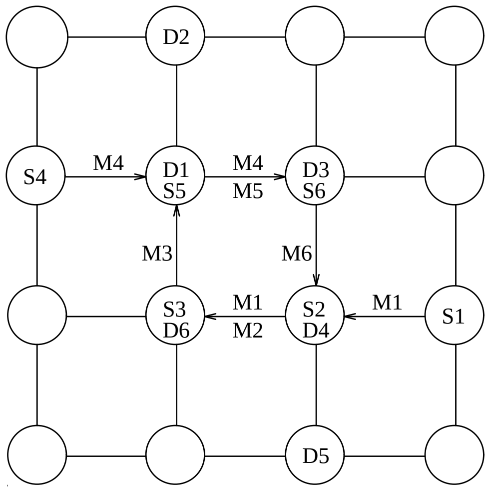


Figure 10: West-First with E and W Virtual Channels

deadlock configuration is shown for each of the five remaining possibilities. The deadlock configurations occur whether or not 0° turns are permitted. **Note**: In some of the examples, deadlock can be avoided by prohibiting additional 90° turns. Such modifications are irrelevant, since the goal is to minimize the number of restrictions.

Lemma 3.5 *Opt-y is not deadlock-free if the routing restrictions shown in Figure 8 are still applied to* C_1*, but the virtual channels are added in the East and West directions.*

Proof. The routing algorithm is fully adaptive only if the 90° turns N–W and S–W using $VC_{2\mathrm{W}}$ are unrestricted. A deadlock configuration is shown in Figure 10. □

Lemma 3.6 *The north-last routing algorithm is not deadlock-free with only six virtual channels and two prohibited* 90° *turns.*

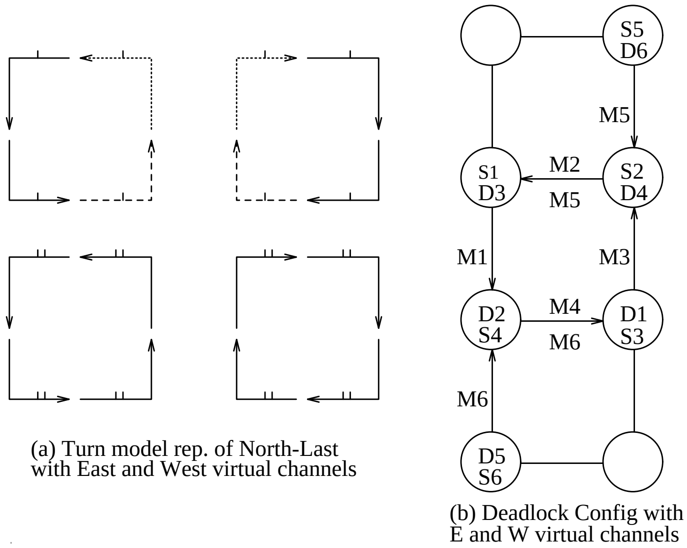
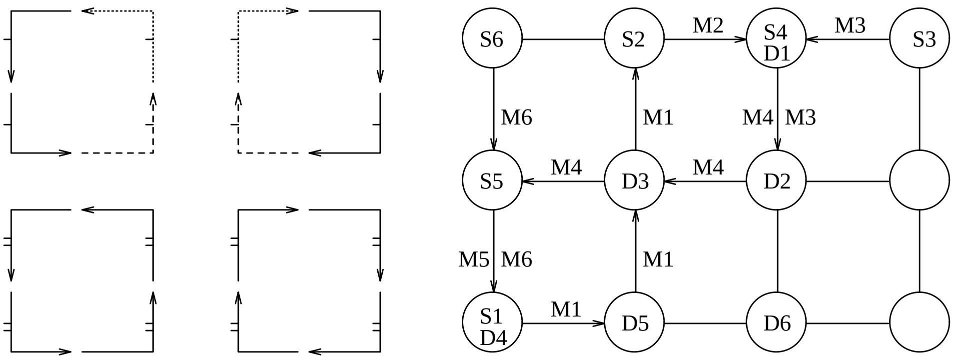


Figure 11: North-Last with E and W Virtual Channels

Proof. The turn model representation of *north-last* with the second virtual channel in the East and West directions is shown in Figure 11(a). Only the 90° turns are shown, since the 0° turns are not used for showing deadlock. A deadlock configuration for this routing algorithm is shown in Figure 11(b). The turn model representation with the second virtual channel in the North and South directions is shown in Figure 12(a). A deadlock configuration for this routing algorithm is shown in Figure 12(b). □

Lemma 3.7 *The negative-first routing algorithm is not deadlock-free with only six virtual channels and two prohibited* 90° *turns.*

Proof. The turn model representation of *negative-first* with the second virtual channel in the East and West directions is shown in Figure 13(a). Again, only the 90° turns are shown, since the 0° turns are not used for showing deadlock. A deadlock



(a) Turn model rep. of North-Last with North and South virtual channels

(b) Deadlock Config with N and S virtual channels

Figure 12: North-Last with N and S Virtual Channels

configuration for this routing algorithm is shown in Figure 13(b). The turn model representation with the second virtual channel in the North and South directions is shown in Figure 14(a). A deadlock configuration for this routing algorithm is shown in Figure 14(b). □

The only remaining possibility is to divide the prohibited turns between the virtual channels so that one is applied to VC_1 and the other to VC_2. The deadlock configurations shown in Figures 10 – 14 are still possible with at most minor modifications. This leads to theorem 3.4, which is proved for *symmetric* routing algorithms. Symmetry is preserved if the virtual channels are added in the X dimension with a corresponding set of restrictions and/or the restrictions are applied to VC_2 instead of VC_1.

Theorem 3.4 *Ignoring symmetry, opt-y prohibits fewer* $90°$ *turns than any other deadlock-free fully adaptive routing algorithm that uses only six virtual channels.*

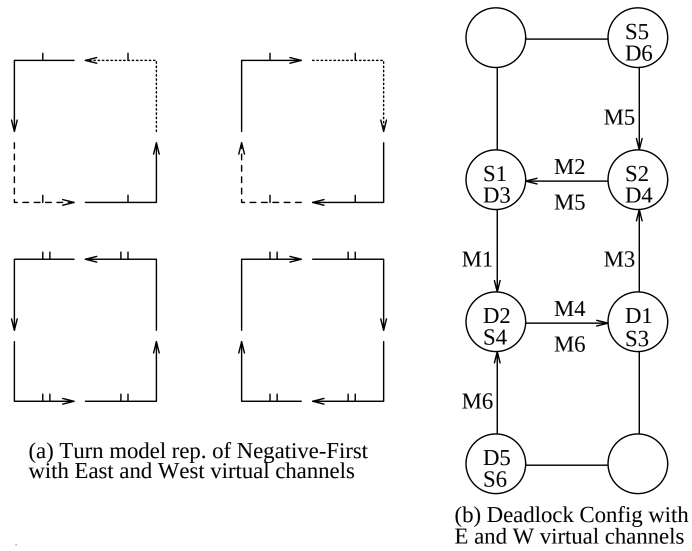


Figure 13: Negative-First with E and W Virtual Channels

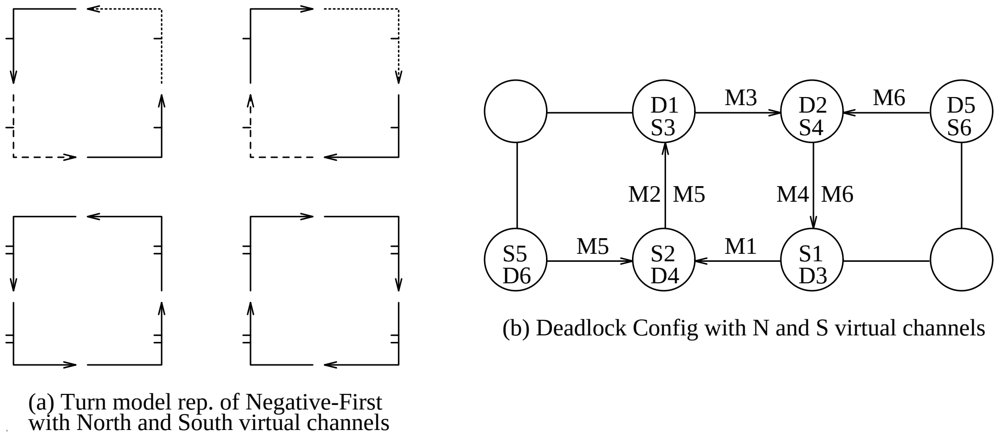


Figure 14: Negative-First with N and S Virtual Channels

Proof. By lemma 3.4, at least two 90° turns must be prohibited. Ignoring symmetry, there are only three such routing algorithms [39]. Opt-y uses one of these, however, by lemma 3.5, it is not deadlock-free if the virtual channels are added in the East and West directions instead of the North and South directions. The other possibilities, *north-last* and *negative-first*, are both not deadlock-free (by lemma 3.6 and lemma 3.7, respectively). Since there are no other possibilities to consider, opt-y is the only deadlock-free fully adaptive routing algorithm that requires only six virtual channels and prohibits only two 90° turns. □

Theorem 3.5 *No routing algorithm using only six virtual channels per router can have fewer restrictions than those imposed by opt-y.*

Proof. By lemma 3.4 and theorem 3.4, it is clear that only the 0° turns need to be considered. Opt-y restricts the use of the 0° turns from $VC_{2\mathrm{N}}$ to $VC_{1\mathrm{N}}$ and from $VC_{2\mathrm{S}}$ to $VC_{1\mathrm{S}}$ to messages that do not need to route further West. Figure 15 gives an example where deadlock occurs when the restriction on the 0° turn from $VC_{2\mathrm{N}}$ to $VC_{1\mathrm{N}}$ is removed. A similar deadlock configuration can be constructed if the restriction on the 0° turn from $VC_{2\mathrm{S}}$ to $VC_{1\mathrm{S}}$ is removed. Since the restrictions on the other two 0° turns are a direct consequence of the prohibited 90° turns, opt-y cannot be made more adaptive. □

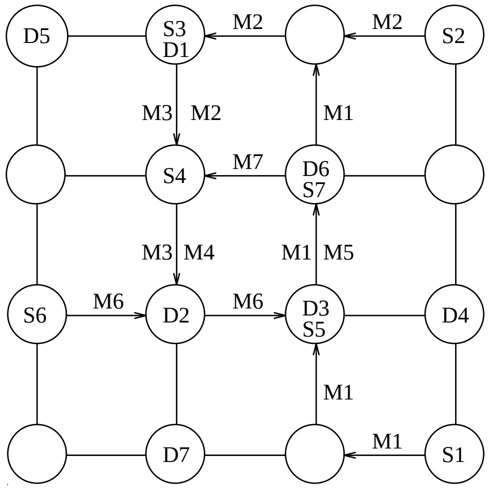


Figure 15: Deadlock due to Fewer Restrictions

3.5 Comparison of Opt-y with Mad-y

Opt-y is compared with mad-y, the least restrictive of any previously proposed routing algorithm. The comparison is limited to minimal routing. The virtual channel requirements are the same for both routing algorithms. The turn model representation of mad-y is shown in Figure 16. Mad-y has the following two sets of constraints:

- Four 90° turns, N–W using $VC_{2\mathrm{N}}$, S–W using $VC_{2\mathrm{S}}$, E–N using $VC_{1\mathrm{N}}$, and E–S using $VC_{1\mathrm{S}}$, are prohibited.
- The 0° turns from $VC_{2\mathrm{N}}$ to $VC_{1\mathrm{N}}$ and from $VC_{2\mathrm{S}}$ to $VC_{1\mathrm{S}}$ are prohibited.

The following four restrictions arise solely from the previous constraints:

- The 0° turns from $VC_{1\mathrm{N}}$ to $VC_{2\mathrm{N}}$ and from $VC_{1\mathrm{S}}$ to $VC_{2\mathrm{S}}$ are allowed only when a message does not need to route further West and has not already routed East.

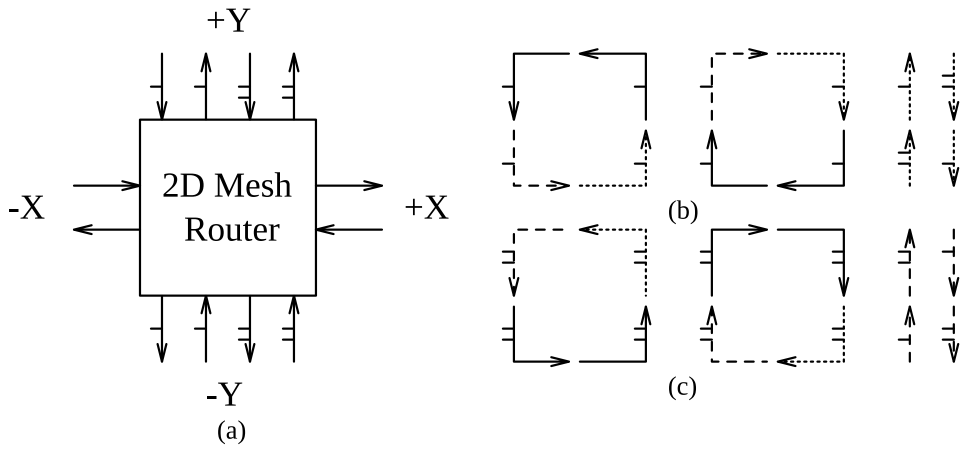


Figure 16: Maximally Fully Adaptive Routing

The West-bound restriction exists because a message cannot route West from $VC_{2\mathrm{N}}$ or $VC_{2\mathrm{S}}$. The East-bound restriction exists because a message never uses $VC_{1\mathrm{N}}$ or $VC_{1\mathrm{S}}$ after routing East.

- Two 90° turns, S–E using $VC_{1\mathrm{S}}$ and N–E using $VC_{1\mathrm{N}}$, are allowed only when the message has not routed East. These turns are restricted only because a message cannot use $VC_{1\mathrm{N}}$ or $VC_{1\mathrm{S}}$ after VC_{E}.

- Two 90° turns, W–S using $VC_{2\mathrm{S}}$ and W–N using $VC_{2\mathrm{N}}$, are allowed only when the message is not routed further West. These turns are restricted only because the turns from $VC_{2\mathrm{N}}$ or $VC_{2\mathrm{S}}$ to VC_{W} are prohibited.

- $VC_{2\mathrm{N}}$ and $VC_{2\mathrm{S}}$ cannot be used by the router at the source if the message needs to route West.

Mad-y allows four more 90° turns than double-y, but a message can use these additional turns at most once. Furthermore, a message can make at most one 0° turn, since a message cannot use $VC_{1\mathrm{N}}$ or $VC_{1\mathrm{S}}$ after switching to $VC_{2\mathrm{N}}$ or $VC_{2\mathrm{S}}$. Finally, a message cannot make both a restricted 90° turn and a 0° turn. A message that makes a restricted W–N or W–S turn is no longer in $VC_{1\mathrm{N}}$ or $VC_{1\mathrm{S}}$, so it cannot make a 0° turn. Similarly, a message that makes a restricted N–E or S–E turn never uses $VC_{1\mathrm{N}}$ or $VC_{1\mathrm{S}}$ again, so it cannot make a 0° turn. A message that makes a 0° turn cannot make a restricted turn to the East, because the message never uses $VC_{1\mathrm{N}}$ or $VC_{1\mathrm{S}}$ again. Similarly, a message can make a 0° turn only after routing completely in the West direction, so it cannot later make a restricted turn from the West direction.

Opt-y prohibits only two 90° turns and restricts only two 90° turns. Mad-y prohibits four 90° turns and restricts four 90° turns. Mad-y also prohibits half the 0° turns and restricts the other half. For opt-y, all the 0° turns are restricted, so none of the 0° turns are prohibited.

Opt-y also makes much better use of the restricted turns. The 0° turns are restricted to messages that no longer need to route West, but messages can switch from either virtual channel to the other. This allows a message to make a 0° turn *more than once*. A message also has the possibility of making a restricted 90° turn *and* some 0° turns.

Not only does opt-y have fewer restrictions, but the router control logic is also simpler. Mad-y has a routing relation of the form $R : C \times N \times N \to C^p$, so the set of output channels from which a message can choose depends not only on the current node and destination but also on the input channel. Opt-y does not differentiate based on the input channel, because the routing relation is of the form $R : N \times N \rightarrow C^p$. This simplifies the hardware required to choose the output channel on which to route the message, since the choice is independent of the input channel.

3.6 Generalization

The Optimal Fully Adaptive routing algorithm can be extended to n-dimensional meshes in a straightforward manner. Minimal routing is still assumed. The routing algorithm needs only the minimum number of virtual channels and imposes only the minimum number of routing restrictions needed to avoid deadlock. Two different generalizations of opt-y are provided. The first uses Duato's proof technique to prove the lower bound on the number of virtual channels. A slightly different generalization, however, is required to prove the minimum number of routing restrictions. The routing algorithm is first generalized using the following steps:

- Assign a channel to both directions of each dimension.
- Number the dimensions in some order and add a second virtual channel to both directions of all dimensions except the first dimension.
- Allow a message to route along the second virtual channel at any time.
- For each dimension except the last, select one of the two directions as the chosen direction of that dimension. Prohibit a message from routing on the first virtual

channel of any direction until it has completed routing in the chosen direction of *all* lower dimensions.

- Allow a message to make a 0° turn between the two virtual channels of a direction only after the message has completed routing in the chosen direction of *all* lower dimensions.

For example, consider a 4D mesh with dimensions XYZW and number the dimensions 1, 2, 3, and 4, respectively. The positive and negative directions in the Z dimension are called Up and Down, respectively. The positive and negative directions in the W dimension are called In and Out, respectively. Let the negative directions in the X, Y, and Z dimensions be the chosen directions. A second virtual channel is added in both directions of the Y, Z, and W dimensions. A message can route adaptively in any dimension using virtual channel one, once it has completed routing in the chosen direction of all lower dimensions. For example, a message can use $VC_{1\mathrm{N}}$ and $VC_{1\mathrm{S}}$ only after it has completed routing West. Similarly, a message can use $VC_{1\mathrm{U}}$ and $VC_{1\mathrm{D}}$ only after it has completed routing West and South and a message can use $VC_{1\mathrm{I}}$ and $VC_{1\mathrm{O}}$ only after it has completed routing West, South, and Down. A message can route, without any restrictions, in the X dimension and in any direction using virtual channel two.

A block diagram of the YZ plane of an n-dimensional router is shown in Figure 17(a). The virtual channels are differentiated by marking the first virtual channel in each direction with a single dash and the second virtual channel with a double dash. This configuration has thirty-two 90° turns and eight 0° turns. As before, *unrestricted* turns are indicated by solid lines, *restricted* turns by dashed lines, and *prohibited* turns by dotted lines. For simplicity, only 90° turns are shown, since all 0° turns are restricted. The YZ plane of the Optimal Fully Adaptive routing algorithm has the following two sets of constraints (See Figure 17(b)):

- Four of the thirty-two 90° turns, namely U–S from VC_{1U} to VC_{1S} or VC_{2S} and D–S from VC_{1D} to VC_{1S} or VC_{2S}, are prohibited.
- The 0° turns from VC_{2U} to VC_{1U} and from VC_{2D} to VC_{1D} are not allowed when the message needs to route further West or South.

The following restrictions arise solely from the previous constraints:

- The four 90° turns S–U from VC_{1S} or VC_{2S} to VC_{1U} and S–D from VC_{1S} or VC_{2S} to VC_{1D} are not allowed when the message needs to route further West or South. These turns are restricted only because the U–S turns from VC_{1U} and D–S turns from VC_{1D} are prohibited.
- The 0° turns from VC_{1U} to VC_{2U} and from VC_{1D} to VC_{2D} are not allowed when the message needs to route further West or South, because VC_{1U} and VC_{1D} are not used until the message has completed routing West and South.
- VC_{1U} and VC_{1D} cannot be used by the router at the source if the message needs to route West or South.

Theorem 3.6 *The Optimal Fully Adaptive routing algorithm is fully adaptive for n-dimensional meshes.*

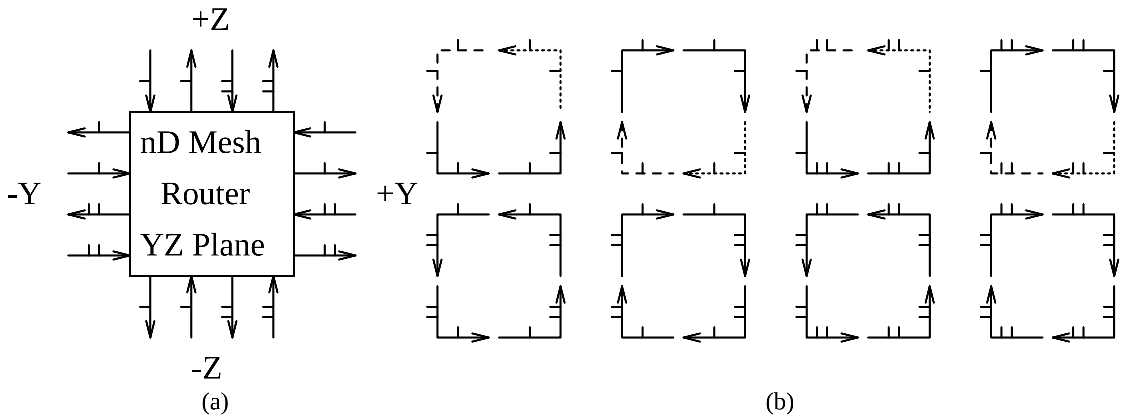


Figure 17: YZ Plane of an n-dimensional Router

Proof. Messages can use the second virtual channel at any time. Messages can also route in either direction of the lowest dimension at any time, since there are no restrictions on using the virtual channels in the lowest dimension. This allows the message to route in any direction that moves the message closer to the destination. □

Since the use of the first virtual channels depends on in which of the chosen directions the message still needs to route, the routing relation is of the form $R : N \times N \to C^p$. The proof that the routing algorithm is deadlock-free relies on Duato's proof of a sufficient condition for deadlock freedom [23]. C_1 consists of only virtual channel one in each direction, so R_1 uses only the permitted 90° turns among the first virtual channels.

Lemma 3.8 *Routing algorithm R_1 is connected.*

Proof. A message can reach any destination from any source. R_1 prevents only a channel in a higher dimension from using a channel in a lower chosen direction, hence dimension-order routing is a subset of R_1. The proof follows immediately from the fact that dimension-order routing is connected. □

Lemma 3.9 *Routing algorithm R_1 is acyclic and therefore deadlock-free.*

Proof. Since 180° turns are not permitted for minimal routing, any cycle requires that there be dependencies across multiple dimensions. In addition, *both* directions of each dimension in the cycle must be used to form the cycle. For a cycle to exist, there must be a dependency from some higher dimension(s) of the cycle to both directions in the lowest dimension of the cycle. R_1, however, requires that the chosen direction of each dimension be used before any direction of a higher dimension, so no cycle is possible. □

Lemma 3.10 *The additional channels of routing algorithm R do not introduce cycles in the extended channel dependency graph of C_1.*

Proof. The only possibility for a cycle in C_1 is for an *indirect* dependency to be introduced by using one or more of the second virtual channels. First, note that R_1 prevents only a channel in a higher dimension from using a channel in a lower chosen direction. Indirect dependencies that allow a use of the channels different from R_1 could arise in only three ways: (1) A message uses virtual channel one in a chosen direction after using virtual channel one in a higher dimension. This cannot occur, since the routing algorithm, R, explicitly prohibits the use of the virtual channels in this order. (2) A message uses virtual channel one in some chosen direction and later uses virtual channel one in another chosen direction. This is a special case of (1) if the first chosen direction is in a higher dimension than the second chosen direction. The routing algorithm does not allow this possibility. If the first chosen direction is in a lower dimension than the second chosen direction, then this possibility does not cause a problem, since this is allowed by R_1. (Note that this turn is allowed by R_1 and R only when the message has completed routing in the lower of the two dimensions.) (3) A message uses virtual channel one in a chosen direction, routes in another dimension, and then uses virtual channel one in the same chosen direction. Routing algorithm R allows this possibility, but it does not cause a deadlock. The channel dependency graph for C_1 has no dependencies from a channel in a higher dimension to the channel in the chosen direction of a lower dimension. There are also no dependencies from this channel to the channel in any chosen direction in a lower dimension. Therefore, these indirect dependencies create new dependencies only among channels in this chosen direction and channels in unchosen directions of lower dimensions, however, these dependencies do not create a cycle. Since a mesh has no wrap-around channels and minimal routing is used, there are no cycles among channels in the same chosen direction and R_1 already allows the use of a channel in an unchosen direction after the use of a channel in a higher chosen direction. Hence the extended channel dependency graph of C_1 is acyclic. □

Theorem 3.7 *The Optimal Fully Adaptive routing algorithm is deadlock-free for n-dimensional meshes.*

Proof. The proof follows immediately from lemmas 3.8 – 3.10. □

Theorem 3.8 *The Optimal Fully Adaptive routing algorithm uses only the minimum number of virtual channels for deadlock-free fully adaptive wormhole routing on a mesh.*

Proof. Assume that a virtual channel is removed. Removing the virtual channel in either direction of the lowest dimension makes the mesh disconnected. If the virtual channel is removed from one of the other dimensions, then there are only five virtual channels in the plane formed by this dimension and the lowest dimension. It is possible to generate a set of messages that need to route in only these two dimensions. By theorem 3.3 it is known that this configuration is not both fully adaptive and deadlock-free. □

By theorem 3.8, the Optimal Fully Adaptive routing algorithm requires only the minimum number of virtual channels for deadlock-free fully adaptive routing. The remaining proofs are used to show that the Optimal Fully Adaptive routing algorithm has the fewest restrictions of any fully adaptive routing algorithm with $4n-2$ virtual channels, and that, ignoring symmetry, all other fully adaptive routing algorithms have more restrictions. The 2D case has been used to prove optimality when there are six virtual channels between the two dimensions, which occurs with the lowest dimension and any other dimension. There are eight virtual channels, two virtual channels per physical channel, for every pair of dimensions, except when the lowest dimension is one of these two dimensions. The following lemmas are used to prove that the routing algorithm is optimal with two virtual channels in all four directions.

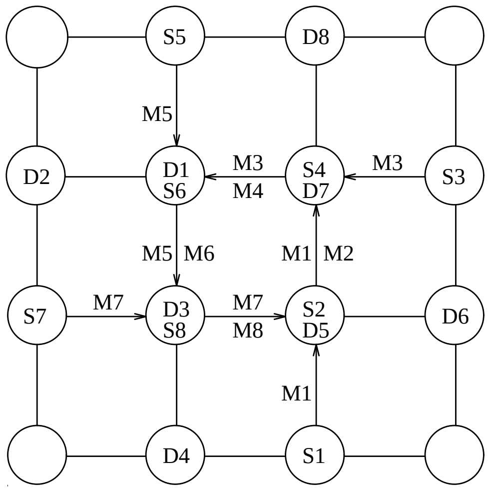


Figure 18: Deadlock with 2 VCs per Physical Channel

Lemma 3.11 *At least four* 90° *turns must be prohibited to make a routing algorithm deadlock-free when two virtual channels are used for each physical channel.*

Proof. Figure 18 shows a deadlock configuration that arises when no 90° turns are prohibited. A similar deadlock configuration can be constructed for a set of messages routing in the clockwise direction. The only way to prevent each deadlock configuration is to prohibit at least one *channel* from being used before *any* channel in at least one other direction. Otherwise, the deadlock configuration shown in Figure 18 cannot be avoided. Each message could still use the virtual channel(s) shown in Figure 18 and wait for at least one of the two channels in the other direction. There are two virtual channels in each direction, so at least two 90° turns must be prohibited for each of the two orientations (CW and CCW). Therefore, a total of at least four 90° turns must be prohibited. □

By lemma 3.11, at least four 90° turns must be prohibited between each pair of dimensions. Also, the prohibited turns must be from one virtual channel to either virtual channel in another direction. The Optimal Fully Adaptive routing algorithm, which uses an extension of *west-first*, prohibits exactly four 90° turns. Hence, ignoring symmetry, the only possibilities to consider are extensions[2] of *north-last* and *negative-first*. A deadlock configuration is shown for these two possibilities when only four 90° turns are prohibited. Both deadlock configurations occur whether or not 0° turns are permitted. Note that for clarity of exposition, we use directions such as North and West when discussing one plane of the n-dimensional mesh, even though the plane could be formed by any two dimensions.

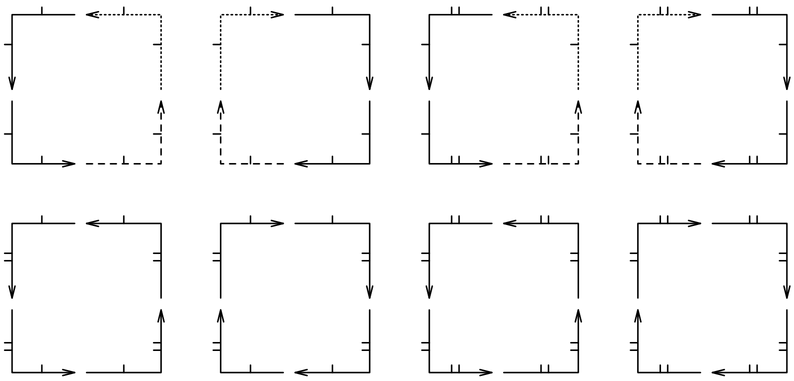
Lemma 3.12 *The north-last routing algorithm is not deadlock-free with two virtual channels per physical channel and four prohibited* 90° *turns.*

Proof. The turn model representation of *north-last* with two virtual channels per physical channel is shown in Figure 19(a). Only the 90° turns are shown, since deadlock occurs with or without 0° turns. A deadlock configuration is shown in Figure 19(b). □

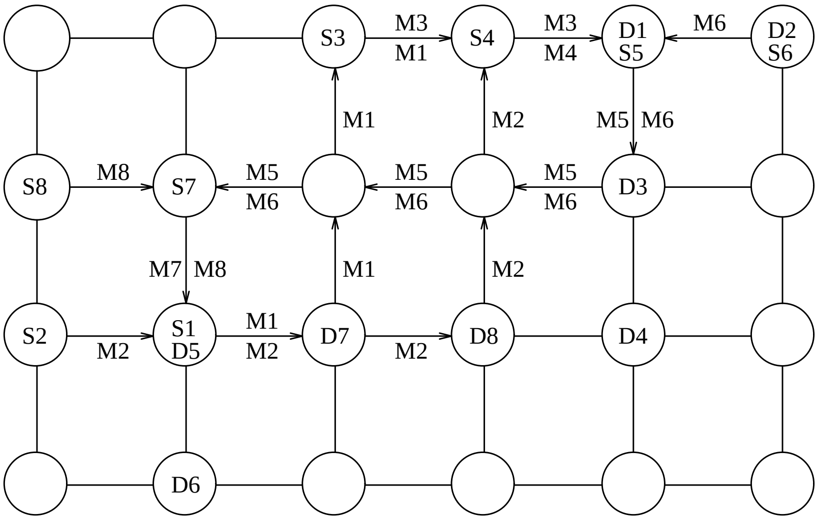
Lemma 3.13 *The negative-first routing algorithm is not deadlock-free with two virtual channels per physical channel and four prohibited* 90° *turns.*

Proof. The turn model representation of *negative-first* with two virtual channels per physical channel is shown in Figure 20(a). Only the 90° turns are shown, since

[2] These extensions have two virtual channels per physical channel and impose additional restrictions on the second set of virtual channels to satisfy lemma 3.11.



(a) Turn model rep. of North-Last with
two virtual channels per physical channel



(b) Deadlock Config with two virtual channels per physical channel

Figure 19: North-Last with 2 VCs per Physical Channel

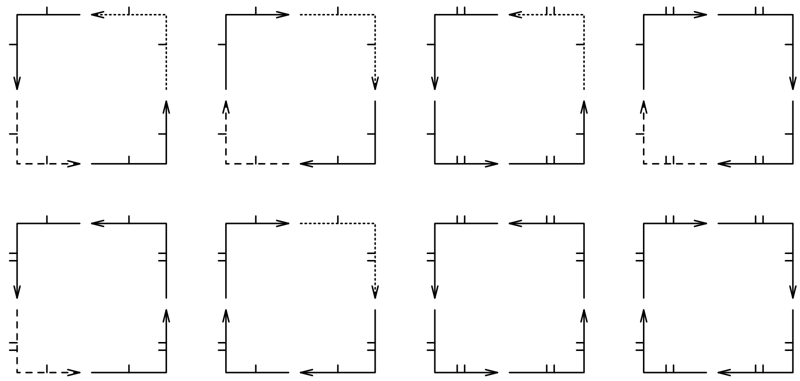
deadlock occurs with or without 0° turns. A deadlock configuration is shown in Figure 20(b). □

Theorem 3.9 *Ignoring symmetry, the Optimal Fully Adaptive routing algorithm prohibits fewer 90° turns than any other fully adaptive deadlock-free routing algorithm with two virtual channels per physical channel.*

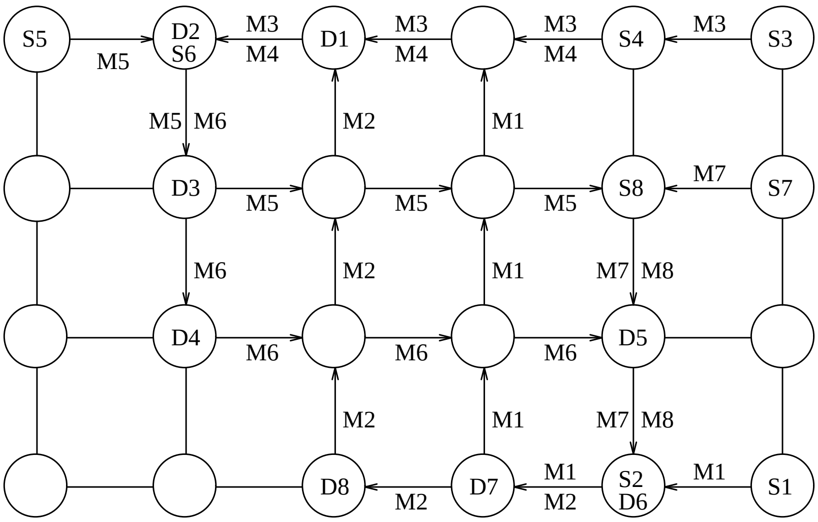
Proof. By lemma 3.11, at least four 90° turns must be prohibited. Ignoring symmetry, there are only three routing algorithms to consider. The Optimal Fully Adaptive routing algorithm is one of these. By lemma 3.12 and lemma 3.13, the other two possibilities are not deadlock-free. Therefore, the Optimal Fully Adaptive routing algorithm is the only deadlock-free fully adaptive routing algorithm with two virtual channels per physical channel that prohibits only four 90° turns. □

Theorem 3.10 *No routing algorithm using two virtual channels per physical channel can have fewer restrictions than those imposed by the Optimal Fully Adaptive routing algorithm.*

Proof. By lemma 3.11 and theorem 3.9, it is clear that only the 0° turns need to be considered. The Optimal Fully Adaptive routing algorithm restricts the 0° turns from the second virtual channel to the first virtual channel in the higher dimension to messages that do not need to route further in the chosen direction of the lower dimension. Figure 15 is easily modified to give an example where deadlock occurs when the restriction on the 0° turn in the positive direction of the higher dimension is removed. A similar deadlock configuration can be constructed if the restriction on



(a) Turn model rep. of Negative-First with two virtual channels per physical channel



(b) Deadlock Config with two virtual channels per physical channel

Figure 20: Negative-First with 2 VCs per Physical Channel

the $0°$ turn in the negative direction is removed. Since the restrictions on the other two $0°$ turns (from the first virtual channel to the second) are a direct consequence of the prohibited $90°$ turns, the Optimal Fully Adaptive routing algorithm cannot be made more adaptive. □

The Optimal Fully Adaptive routing algorithm proposed at the beginning of Section 3.6 requires only the minimum number of restrictions for messages that route in any two dimensions. The routing restrictions can be relaxed for messages that traverse more than two dimensions, although the routing restrictions for any 2D plane of the mesh remain unchanged. The relaxed restrictions permit a message to use virtual channel one in any dimension when the message does not need to route in the chosen direction of the *lowest* dimension in which the message needs to route. This allows a message to use the first virtual channel in a higher dimension when the message has not completed routing in the chosen direction of *all* lower dimensions. To maintain deadlock freedom, a message waits for the first virtual channel in only the lowest dimension in which the message still needs to route. We next prove that the routing algorithm is still deadlock-free unless the routing restrictions are relaxed further.

Theorem 3.11 *The Optimal Fully Adaptive routing algorithm is deadlock-free even with the aforementioned relaxation of the routing restrictions.*

Proof. Any cycle requires that there be dependencies across multiple dimensions, with both directions of each dimension in the cycle used to form the cycle. Note that for a deadlock configuration to exist, every message in the cycle must be waiting for a channel held by some other message in the cycle. Without loss of generality, assume that the Y dimension is the lowest dimension in the cycle, with South as the chosen direction. Consequently, no message in the cycle still needs to route in the X dimension. Divide the cycle into two planes along the Y dimension. The cycle includes some message, m_i, that has used at least one channel in the North plane and is either waiting to route South or has routed South. The cycle requires that some other message waits for a channel that m_i has used in the North plane. Otherwise, there is either a message in the North plane that is not blocked, which resolves the cycle, or there is a cycle in just the North plane, which is a contradiction of the assumption that the Y dimension is used to form the cycle. Recall that the routing algorithm does not permit a message to use virtual channel one when the message needs to route in the chosen direction of the lowest dimension in which the message needs to route. Further, a message waits for only the first virtual channel in the lowest dimension in which the message needs to route. Since m_i has not used any virtual channel one in the North plane, m_i cannot block a message in the North plane. Therefore, a cycle is not possible and the routing algorithm is deadlock-free. □

Theorem 3.12 *The Optimal Fully Adaptive routing algorithm imposes the minimum number of restrictions for deadlock freedom on an n-dimensional mesh.*

Proof. By theorems 3.5 and 3.10, the Optimal Fully Adaptive routing algorithm imposes the minimum number of restrictions for each 2D plane of the mesh and precisely this set of restrictions is required. The only routing restriction is that a message cannot use virtual channel one in a higher dimension when the message needs to route is in the chosen direction of the lowest dimension in which the message needs to route. Deadlock can occur if this restriction is relaxed. Without loss of generality, assume that a message, m_1, used $VC_{1\mathrm{U}}$ even though m_1 needs to route South. (Let South be the chosen direction in the Y dimension and assume that m_1 does not need to route in the X dimension.) There are two messages that route South (and block m_1), use virtual channels in the appropriate dimensions, and wait for $VC_{1\mathrm{D}}$ in the same 2D plane where m_1 used $VC_{1\mathrm{U}}$. Use additional messages to form a deadlock configuration similar to the one shown in Figure 18. The routing algorithm is no longer deadlock-free, so the routing restrictions cannot be relaxed. □

3.7 Comparison for Arbitrary Dimension Meshes

Table 2 summarizes the virtual channel requirements per router for some routing algorithms defined for n-dimensional meshes. This is a complete list of the adaptive routing algorithms for n-dimensional meshes that were known when we proposed the Optimal Fully Adaptive routing algorithm. (Other adaptive routing algorithms for n-dimensional meshes have been proposed subsequent to the publication of our routing algorithm, however, these routing algorithms are less adaptive than the Optimal Fully Adaptive routing algorithm.) The partially adaptive routing algorithms proposed by Glass and Ni [39], although requiring no additional virtual channels, allow only about $1/2^{n-1}$ of the paths provided by a fully adaptive routing algorithm for an n-dimensional mesh. Three virtual channels per physical channel are used by Chien and Kim [9] to support partially adaptive routing for arbitrary dimension meshes. Jesshope, Miller, and Yantchev [48] propose a fully adaptive routing algorithm that divides an n-dimensional mesh into 2^n virtual networks and uses n virtual channels for each virtual network. Fully adaptive routing on n-dimensional meshes requires 2^{n-1} subnetworks with $n+1$ levels per subnetwork and one virtual channel per level for each router using the routing algorithm proposed by Linder and Harden [66]. For comparison, Optimal Fully Adaptive routing algorithm requires only $4n-2$ virtual channels for fully adaptive routing on an n-dimensional mesh.

Table 2: Comparison of the Number of Virtual Channels per Router

	Nonadaptive	Partially Adaptive		Fully Adaptive		
Topology	Dimension Order	Glass & Ni	Chien & Kim	Jesshope, Miller, & Yantchev	Linder & Harden	Dally & Aoki
2D Mesh	4	4	6	8	6	8
3D Mesh	6	6	12	24	16	12
4D Mesh	8	8	18	64	40	16
nD Mesh	$2n$	$2n$	$6n-6$	$n2^n$	$(n+1)2^{n-1}$	$4n$

Table 2 illustrates the dramatic reduction in the number of virtual channels required by the Optimal Fully Adaptive routing algorithm compared to the fully adaptive routing algorithms proposed in [48, 66].

The dynamic routing algorithm concurrently proposed by Dally and Aoki [16] requires two additional virtual channels and imposes substantially more routing restrictions on the routing algorithm. For each 2D plane with two virtual channels per physical channel, the Optimal Fully Adaptive routing algorithm prohibits only four of the thirty-two 90° turns and none of the eight 0° turns, compared to the twelve 90° turns and four 0° turns prohibited by the dynamic routing algorithm.

Note that fully adaptive routing is possible using about *two-thirds* the virtual channels needed for the partially adaptive routing algorithm proposed by Chien and Kim [9]. The only adaptive routing algorithms that require fewer virtual channels are the partially adaptive algorithms proposed by Glass and Ni [39], however, with less than twice as many virtual channels, the Optimal Fully Adaptive routing algorithm allows an average of 2^{n-1} times as many paths.

3.8 Extra Virtual Channels

The Optimal Fully Adaptive routing algorithm requires only the minimum number of virtual channels. Previous routing algorithms proposed for n-dimensional meshes require more virtual channels. It may appear that adaptiveness can be increased by using one of these other routing algorithms, however, the Optimal Fully Adaptive routing algorithm can be extended to use additional virtual channels with only the minimum number of routing restrictions.

Consider a router with more than two virtual channels per physical channel. Generalizing lemma 3.11, the routing algorithm is not deadlock-free unless, for each pair of dimensions, one channel in both directions of the higher dimension is not used by messages that need to route in the chosen direction of the lower dimension. These restrictions depend on the ordering of the dimensions. For example, the chosen direction in the lowest dimension (based on the ordering) cannot be used after the first virtual channel in any other dimension.

Minimizing the number of restrictions requires minimizing the number of prohibited 90° turns, since the 0° turns are restricted for all directions except those in the lowest dimension. The first virtual channel of both directions of every other dimension cannot be used by a message that needs to route further in the chosen direction of the lowest dimension, so the direction with the fewest virtual channels should be used as the chosen direction of the lowest dimension. Similarly, the direction (of the remaining dimensions) with the fewest virtual channels should be used as the chosen direction of the next lowest dimension and so on. By selecting the dimension ordering and chosen directions in this way, the number of prohibited 90° turns can be minimized. The number of 90° turns that are prohibited for an n-dimensional mesh is:

$$2\sum_{i=1}^{n-1} VC_i(n-i) \tag{3.1}$$

where VC_i is the number of virtual channels in the chosen direction of dimension i.

A simple algorithm can be used to minimize the number of prohibited 90° turns. The pseudo-code is given below:

/* The array `chosen` holds the chosen directions. */

/* The array `dims` holds the dimension ordering. */

/* The `dimension()` function returns the dimension of a direction. */

/* The boolean array `used` indicates whether or not a dimension has been assigned. */

```
vcs := A list of directions sorted by increasing number of virtual channels;
for i := 0 to n − 1 do
    chosen[i] := {};
    dims[i] := {};
    used[i] := False;
endfor;
i := 0;
while (vcs <> ∅) do
    smallest := pop(vcs);
    if (not used[dimension(smallest)]) then
        chosen[i] := smallest;
        dims[i] := dimension(smallest);
        used[dimension(smallest)] := True;
        i := i + 1;
    endif;
endwhile;
```

For example, consider a 3D mesh with the following number of virtual channels in each direction: East = 2, West = 3, North = 6, South = 5, Up = 5, and Down = 3. There are 24 different ways to order the dimensions and choose the directions, because there are six dimension orderings with two of four directions chosen. The minimum number of prohibited 90° turns is achieved by using a dimension ordering of XZY and making East and Down the chosen directions. This configuration imposes fourteen prohibited 90° turns, since the 90° turns from $VC_{1\mathrm{N}}$, $VC_{1\mathrm{S}}$, $VC_{1\mathrm{U}}$, and $VC_{1\mathrm{D}}$ to either East virtual channel are prohibited and the 90° turns from $VC_{1\mathrm{N}}$ and $VC_{1\mathrm{S}}$ to any of the three Down virtual channels are prohibited. Any other ordering of the dimensions or any different choice of chosen directions requires that more than fourteen 90° turns be prohibited to prevent deadlock.

3.9 Summary

In this chapter, we have presented an optimal fully adaptive routing algorithm for meshes. The routing algorithm is optimal in the number of virtual channels required and the restrictions placed on the use of those channels. The routing algorithm can be extended to support additional virtual channels with the minimum number of restrictions. The proofs of optimality are specific to the mesh, however, the underlying ideas suggest the possibility for extension to other topologies.

The decision was made to extend these proof ideas into a more general framework. In addition, the problem of proving optimality would be greatly simplified by a necessary and sufficient condition for proving deadlock freedom. This would remove the requirement of constructing a deadlock configuration for each of the alternative routing algorithms that are not deadlock-free.

The identification of a necessary and sufficient condition for deadlock-free adaptive routing has remained an important open problem since Duato [23] pointed out that Dally and Seitz's [17] requirement of an acyclic channel dependency graph is necessary for only nonadaptive routing. Our solution of this open problem is presented in Chapter IV.

CHAPTER IV

Necessary and Sufficient Condition for Deadlock-Free Routing

In the previous chapter, we presented an optimal routing algorithm for n-dimensional meshes. The problem of designing optimal routing algorithms would be simplified by knowing exactly what restrictions are required to guarantee deadlock freedom. Hence, a natural question arises: Exactly how restrictive must the routing algorithm be to guarantee deadlock freedom? In other words, what is a necessary and sufficient condition for deadlock-free routing? In this chapter, we answer the question. Besides providing the only known general necessary and sufficient condition for deadlock freedom, the routing algorithms developed using our methodology can be substantially less restrictive than previous routing algorithms. The underlying proof technique for the necessary and sufficient condition introduces a new type of dependency graph, the *channel waiting graph*, which omits most channel dependencies that cannot be used to create a deadlock configuration.

The necessary and sufficient condition can be used for any adaptive or nonadaptive routing algorithm for wormhole routing, as long as the router does not require global information. *The routing can be based on the input channel, current node, destination* *node, and any other information contained in the message header, although we restrict the discussion to the case where only the destination is contained in the message header.* Our methodology can also be generalized to routing algorithms for store-and-forward routing and virtual cut-through. In fact, the proof technique can be applied to any switching technique that relies on routing restrictions to prevent deadlock. The use of the channel waiting graph simplifies the task of proving deadlock freedom.

We first describe the proof technique as it applies to wormhole routing. The application of our proof technique to wormhole routing is to be published in [88]. In Section 4.1, we present definitions used in the proofs and discussion of our technique. In Section 4.2, we briefly explain why livelock and deadlock are independent. We then present a simple sufficient condition for deadlock freedom in Section 4.3. The necessary and sufficient condition is presented in Section 4.4. Our methodology is formalized in Sections 4.5 and 4.6 with a description of the process for distinguishing between False Resource Cycles and True Cycles, and an algorithm for reducing the channel waiting graph to an appropriate subgraph. Two example routing algorithms are presented in Section 4.7. The first routing algorithm is for nonminimal partially adaptive routing on a mesh and the second is for fully adaptive routing on a hypercube. Both routing algorithms are more adaptive than any previously proposed routing algorithm with similar virtual channel requirements. Finally, in Section 4.8, we describe the applicability of our methodology to other switching techniques and describe the changes required to support store-and-forward routing and virtual cut-through. This "universal" result is to be published in [91].

4.1 Definitions

In addition to the assumptions and definitions presented in Section 3.2 the following definitions are used to define our notation and formalize the presentation of our results.

Definition 4.1 Routing algorithm R_A is *prefix-closed* if a path that R_A permits from node n_i to node n_j that routes through node n_k ($n_i \neq n_k$) implies that R_A also permits the partial path from n_i to the first occurrence of n_k on the path from n_i to n_j when n_k is the destination.

Definition 4.2 Routing algorithm R_A is *suffix-closed* if a path that R_A permits from node n_i to node n_j that routes through node n_k implies that R_A also permits the partial path from n_k to n_j when n_k is the source. Note that every routing algorithm with a routing relation of the form $R : N \times N \rightarrow C^p$ is suffix-closed.

Definition 4.3 Routing algorithm R_A is *coherent* if R_A is prefix-closed, suffix-closed, and never routes a message through a node more than once.

Definition 4.4 A *waiting channel* is a channel at the source or an intermediate node for which the message waits when the message is unable to proceed because every channel the message can use is unavailable. A message may have multiple waiting channels at the source or an intermediate node.

Definition 4.5 The *channel waiting graph* (CWG) for a given routing algorithm R_A and interconnection network I is a directed graph, $CWG = G(C, E)$. The vertices of

CWG are the channels of I. The arcs of CWG are pairs of channels, (c_i, c_j), where c_j is a waiting channel for a message that occupies c_i. Formally,

$$\begin{gathered} E = \{(c_i, c_j) |\ \exists n_a, n_b \in N \text{ such that } \{\ldots, c_i, \ldots, c_j, \ldots\} \in R_A(n_a, n_b) \\ \text{and } c_j \text{ is a waiting channel for } R_A(n_a, n_b)\} \end{gathered}$$

Note: There is no requirement that the message waits for c_j *immediately* after using c_i, only that the message is long enough to fill the channel queues from c_i to c_j. Since arbitrary message lengths are assumed, this imposes no restrictions under our system model.

Definition 4.6 Consider a path from a source to a destination. Routing algorithm R_A is *wait-connected* if for every input channel on a path, there exists a waiting channel through which the message can be routed. In other words, regardless of which input channel the message uses (including the injection channel when the message is at the source), there is always an output channel for which the message can wait. Formally,

$$\begin{gathered} \forall n_a, n_b \in N, \forall c_a \in C \text{ such that } \{\ldots, c_a, \ldots\} \in R_A(n_a, n_b) \\ \exists c_b \in C \ni \{\ldots, c_a, c_b, \ldots\} \in R_A(n_a, n_b) \text{ and } c_b \text{ is a waiting channel for } R_A(n_a, n_b) \end{gathered}$$

Note: A message must be able to reach its destination. Hence, a blocked message must wait for at least one output channel. Otherwise, this message is never delivered if it reaches an intermediate node where all the output channels are busy. Therefore, any deadlock-free routing algorithm must be wait-connected.

Definition 4.7 A *configuration* is an assignment of messages to channels. The header and data flits of each message are stored in the channel queues and each channel queue holds flits from at most one message. The leading channel is the channel the message has most recently acquired and its channel queue contains the message header. Any other channels occupied by this message contain only data flits. A configuration is *legal* if each message in the configuration occupies one or more consecutive channels and the message header is stored at the head of the leading channel queue that the message occupies; each message occupies only channels the routing algorithm permits the message to use; and the storage capacity of each occupied channel has not been exceeded.

Definition 4.8 A *deadlock configuration* for routing algorithm R_A on interconnection network I is a non-empty legal configuration consisting of a set of messages, $m_1, m_2, \ldots, m_n, n > 1$, where each message, m_i, in the set has acquired at least one channel. The header flit of m_i has not reached its destination and is unable to proceed because every output channel for m_i is unavailable. Moreover, every waiting channel for m_i is occupied by either data flits of m_i or the header or data flits of another message in the set. The data flits at the head of any other channel queue held by m_i are unable to proceed because the next channel queue occupied by m_i is full. Thus, each message is blocked and must wait for an unavailable waiting channel held by another message in the set. Alternatively, when $n = 1$, m_1 waits for a channel already occupied by itself. The set of messages can be ordered such that:

$$\begin{array}{l} m_i \text{ waits for a channel occupied by } m_{i+1} \forall i < n \text{ and} \\ m_n \text{ waits for a channel occupied by } m_1 \end{array}$$

4.2 Livelock Freedom

As mentioned in Section 1.4, a livelock configuration could arise only with nonminimal routing. Livelock occurs when a message is always routed away from the destination. If the message length exceeds the storage capacity of *all* the channel queues in the network, however, the message finally arrives at an intermediate node where the message already occupies every output channel the message is permitted to use. Hence, a sufficiently long message eventually deadlocks. Livelock could occur when a short message is routed in a cycle of n channels and the message can fit in $n-1$ channel queues. The message could then continually route to the same channel used the previous time through the cycle. By assumption 5, however, this message cannot prevent other messages from using these channels when the channels become free, so this message cannot create a deadlock configuration due solely to the livelock problem. Hence, deadlock freedom can be proved even for routing algorithms that are not livelock-free.

4.3 Sufficient Condition

Most techniques for proving that wormhole routing algorithms are deadlock-free require that the channel dependency graph be acyclic in some manner. The channel dependency graph describes the order in which channels can be *used*. From definition 4.8, however, it is clear that any deadlock configuration is based on the waiting channels, rather than the channels a message could use. (The idea of waiting channels was introduced independently by Lin, McKinley, and Ni [64], however, the methodology presented in this chapter is novel.) The routing algorithm may allow a message to use a channel when the channel is free, even if the message is not permitted to wait for this channel when the channel is busy. This is our motivation for using the CWG, since it ignores dependencies that cannot result in deadlock. Since the channel waiting graph is a subset of the channel dependency graph, requiring an acyclic channel waiting graph is less restrictive than requiring an acyclic channel dependency graph.

Theorem 4.1 *If routing algorithm R_A is wait-connected and the CWG for R_A is acyclic, then R_A is deadlock-free.*

Proof. R_A is wait-connected, so every message always has a waiting channel when all output channels are busy. Assume there is a deadlock configuration involving n messages. If $n = 1$, then there is a cycle in the CWG from a channel to itself, which is not possible since the CWG is acyclic. Otherwise, $(\forall i < n)$ there is an edge in the CWG from every channel occupied by m_i to the channel occupied by m_{i+1} for which m_i is waiting (call this channel c'_{i+1}). There is also an edge in the CWG from every channel occupied by m_n to the channel occupied by m_1 for which m_n is waiting (call this channel c'_1). Hence, there is an edge in the CWG from c'_i to c'_{i+1} $(\forall i < n)$ and from c'_n to c'_1. The CWG for R_A is acyclic, however, so no such set of edges is possible. Therefore, no deadlock configuration exists and R_A is deadlock-free. $\square$

An edge in the CWG requires only the existence of a path from some channel to a waiting channel. The specific intermediate channels used between this channel and

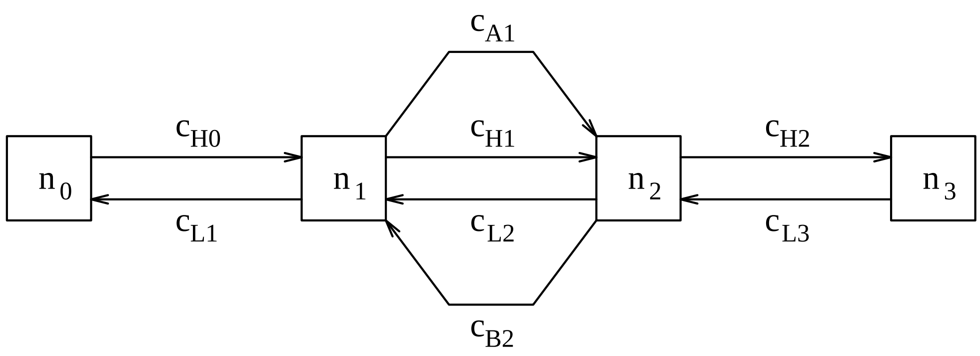


Figure 21: Duato's Example of an Incoherent Routing Algorithm

the waiting channel are not considered when creating the CWG. It is possible that a cycle in the CWG exists only if two or more messages occupy the same channel. For this reason, we divide cycles in the CWG into two classes: False Resource Cycles and True Cycles. A False Resource Cycle is a cycle in the CWG that *requires* at least one channel to be occupied simultaneously by more than one message in order to create the cycle. Note that this shared channel is not necessarily within the cycle. Obviously, a False Resource Cycle cannot occur, since this is physically impossible. Even though the configuration is legal, it is not a reachable configuration. Therefore, a False Resource Cycle cannot be used to create a deadlock configuration. A True Cycle is a cycle in the CWG that permits every message in the cycle to occupy different channels. In Section 4.5, we provide a more complete description of False Resource Cycles and present a technique for distinguishing between False Resource Cycles and True Cycles.

To illustrate the difference between False Resource Cycles and True Cycles, Duato's example [24] of an incoherent routing algorithm is presented. The processors and channels are shown in Figure 21. The routing algorithm permits only minimal routing, with the exception of channel c_{B2}. Channel c_{B2} can be used by a message destined for only node n_3. Clearly, this routing algorithm is not coherent, since a message from n_2 to n_3 can be routed through n_1 using channel c_{B2}, however, a message from n_2 to n_1 cannot use channel c_{B2}.

The CWG for this routing algorithm has a False Resource Cycle and a True Cycle. A message whose input channel is c_{B2} can wait for c_{A1} or c_{H1}. If the message waits for c_{A1}, there is a True Cycle from c_{A1} to c_{A1} that uses c_{B2}. Otherwise, the True Cycle is a cycle from c_{H1} to c_{H1} that uses c_{B2}. There is a False Resource Cycle that involves two messages. A message that occupies c_{A1} and c_{B2} and waits for c_{H1} and a message that occupies c_{H1} and c_{B2} and waits for c_{A1}. Obviously, this False Resource Cycle exists only because both messages simultaneously occupy c_{B2}, which is impossible.

4.4 Necessary and Sufficient Condition

A message is unable to proceed when all output channels the message is permitted to use are busy. This situation can be resolved in one of two different ways: (1) The message could wait for a specific output channel to become free or (2) The message could wait until *any* permitted output channel becomes free. For case (1), the routing algorithm cannot choose to wait for an arbitrary channel, but must choose a channel for which waiting is permitted. Although it is possible that the message has more than one waiting channel, once a waiting channel is chosen, the message must wait for that specific channel to become free. For case (2), the message also has the possibility of waiting on just a subset of more than one output channel. In fact, case (2) includes any routing algorithm that does not conform to case (1). That is, any routing algorithm that does not select a specific waiting channel and wait for that channel until it becomes free. We first prove a necessary and sufficient condition for routing algorithms that belong to case (1), followed by a necessary and sufficient condition for routing algorithms that belong to case (2).

Theorem 4.2 *A routing algorithm, R_A, that requires a message to wait for a specific output channel is deadlock-free iff R_A is wait-connected and the CWG for R_A has no True Cycles.*

Proof. First note that R_A is wait-connected by definition. By theorem 4.1, an acyclic CWG is a sufficient condition for deadlock freedom. A False Resource Cycle cannot result in deadlock, so any False Resource Cycles can be ignored. Since there are no True Cycles, the routing algorithm is deadlock-free.

To prove necessity, assume that a True Cycle with n messages exists. A deadlock configuration can be created from this True Cycle. For each $i < n$, allow message m_i to occupy channel c'_i, some additional channels if necessary, and then wait to acquire channel c'_{i+1} occupied by message m_{i+1}. (Assume that m_i and c'_i are defined as before.) Similarly, message m_n occupies channel c'_n and waits for channel c'_1. Since each message, m_i, in the cycle occupies different channels, it is possible to generate a set of messages that are able to occupy the appropriate channel(s) and then wait for the appropriate channel. To force m_i to wait for the appropriate channel, it is necessary to guarantee that every output channel m_i could use at this router is busy. For any output channel available to m_i that is also available to the source, assume the source has injected a message that is occupying this channel. If R_A is not suffix-closed, however, it is possible that some of the output channels available to m_i can be used only by messages arriving on the input channel used by m_i. For these output channels, assume that a previous message, m_j, used this input channel and was forwarded on one of the output channels. In addition, the length of m_j is assumed to be short enough that it releases the input channel that m_i uses, however, m_j is long enough that it occupies the output channel at this node. By assumption 2, m_j is not necessarily removed from the network immediately, so it is possible that m_j occupies this output channel for a short amount of time. Hence, it is always possible to force m_i to wait for c'_{i+1}. Clearly, each message in the set is waiting for a channel occupied by another message in the set and none of the messages can make progress. Therefore, a deadlock configuration can always be constructed from a True Cycle. $\Box$

For routing algorithms that permit a message to wait for *any* of the output channels to become free, an acyclic CWG is not a necessary condition. Since a blocked message may have multiple waiting channels, messages may be able to avoid channels that form cycles in the CWG by using an alternative channel that is not part of a cycle. Deadlock can be avoided, however, only if at least one of the waiting channels is *guaranteed* to become free. For this reason, we selectively remove edges from the CWG to resolve all True Cycles, as long as the routing algorithm for the resulting graph, CWG', remains wait-connected. We next prove that if no such CWG' exists, then the routing algorithm is not deadlock-free. If such a CWG' does exist, however, then the following theorem can be used to prove deadlock freedom:

Theorem 4.3 *A routing algorithm, R_A, that permits a message to wait for any output channel is deadlock-free iff R_A is wait-connected for some CWG' and this CWG' has no True Cycles.*

Proof. If R_A is wait-connected for the CWG and the CWG has no True Cycles, then the result follows immediately from theorem 4.2, with $CWG = CWG'$. Assume the CWG contains True Cycles. In this case, R_A must be wait-connected for some CWG' without True Cycles.

We first prove sufficiency. Consider a potential deadlock configuration for R_A, involving a cycle of n messages ($n > 0$). This requires that every message in the configuration is waiting for a channel occupied by itself or another message in the cycle. Actually, the message waits for multiple output channels. Since R_A is wait-connected for CWG', at least one of the waiting channels for each message is in CWG'. An output channel in CWG' eventually becomes free and some message in the set is forwarded. However, there is no guarantee that the output channel the message, m_i, eventually acquires is a channel in CWG'. (It is possible that m_i is forwarded along a different channel before an output channel in CWG' becomes free.) If m_i has reached its destination, then the cycle has been resolved. Otherwise, whether or not m_i acquires a channel in CWG', m_i can acquire an output channel in CWG' at the next router because R_A is wait-connected for CWG'. Hence, one of the messages can always be routed and a deadlock configuration cannot occur.

We now prove necessity by showing that the routing algorithm is not deadlock-free if there is no wait-connected CWG' without True Cycles. Assume no wait-connected CWG' without True Cycles exists. Hence, it is possible to generate a set of messages that have no waiting channel that is guaranteed to become available. Furthermore, these messages are all blocking each other, since otherwise it would be possible to guarantee that a waiting channel does become free.

In fact, if even *one* of these messages, m_i, has a waiting channel that becomes free, then either *all* the messages do or the remaining messages in the set (without m_i) satisfy our requirement of a deadlock configuration. Once m_i is delivered, m_{i+1}, the message blocked by m_i, can proceed, since m_{i+1} was waiting for a channel occupied by m_i. Similarly for the remaining messages in the cycle. If a subset of these messages are blocked, this subset can also be resolved if even one has a waiting channel. If this is the case for any potential cycle, however, then, contrary to the assumption, a wait-connected CWG' exists. Therefore, a wait-connected CWG' without True Cycles must exist. $\Box$

By using the necessary and sufficient condition just proposed, it is possible to prove that the incoherent routing algorithm previously discussed is deadlock-free. The routing algorithm is not deadlock-free, however, if a message waits for a specific channel to become free. Consider two messages, one from n_0 to n_3 and one from n_1 to n_3. Assume the message from n_1 to n_3 occupies c_{H1} and c_{H2} and the message from n_0 to n_3 occupies c_{H0}, c_{A1}, and c_{B2}. If this second message waits for c_{A1}, then a deadlock configuration occurs. If this message instead waits for c_{H1}, then a deadlock

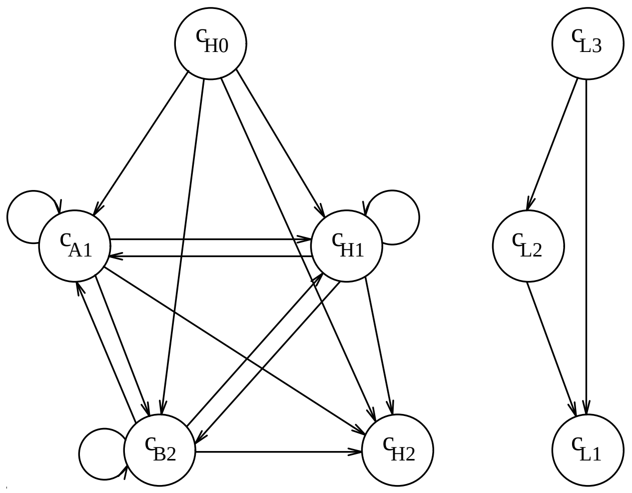


Figure 22: The CWG for the Incoherent Routing Algorithm

configuration occurs when the first message occupies c_{A1} instead of c_{H1} and the second message occupies c_{H1} instead of c_{A1}. Since the routing algorithm does not know which channels have already been used by a message when selecting a waiting channel, the routing algorithm is not deadlock-free if the message waits for a specific channel. On the other hand, if the message waits for both c_{A1} and c_{H1}, then by theorem 4.3 the routing algorithm is deadlock-free. Figure 22 depicts the CWG for this routing algorithm. As discussed previously, there are True Cycles and a False Resource Cycle in this channel waiting graph.

The CWG has True Cycles, however it is possible to create a CWG' with no True Cycles since the routing algorithm is deadlock-free. Edges are removed from the CWG to create CWG' using the following observations:

- Since any message waiting for c_{H2} is guaranteed to reach n_3 and both c_{B2} and c_{H2} can be used only for messages whose destination is n_3, no message is required to wait for c_{B2}. Therefore, the routing algorithm remains wait-connected for CWG' even if all edges to c_{B2} are removed from the CWG.

- A message that arrives on c_{B2} while occupying c_{A1} eventually acquires c_{H1}, since the message occupying c_{H1} is destined for either n_2 or n_3. If the destination is n_2, then the message has reached its destination and c_{H1} will become free. If the destination is n_3, then the message occupying c_{H1} is either occupying c_{H2} or waiting for c_{H2}. Since any message waiting for c_{H2} eventually reaches n_3, c_{H1} eventually becomes free. Similarly, a message that arrives on c_{B2} while occupying c_{H1} eventually acquires c_{A1}. Therefore, the edge from c_{A1} to c_{A1} and the edge from c_{H1} to c_{H1} can be removed from CWG' to create a new CWG' and the routing algorithm is still wait-connected for this new CWG'.

Figure 23 depicts CWG' after these two modifications have been made. There are no True Cycles in CWG'. The only cycle in CWG' is a False Resource Cycle from c_{A1} to c_{H1} and back to c_{A1}, which requires that both messages in the cycle occupy c_{B2}. Since R_A is wait-connected for CWG', deadlock freedom follows immediately from theorem 4.3.

4.5 False Resource Cycles

The CWG is a static graph, however, the dependencies that arise among the channels are dynamic. False Resource Cycles capture this notion of dynamic dependencies among the channels. *When two edges in the CWG both require the use of a common channel, then these two dependencies cannot occur simultaneously.* Cycles in the

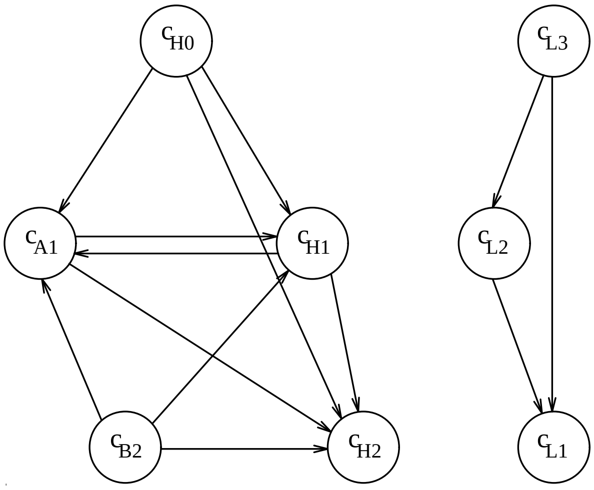


Figure 23: The Final CWG' for the Incoherent Routing Algorithm

CWG that are formed from such dependencies cannot occur in reality, and hence, cannot lead to a deadlock configuration.

4.5.1 Example with Minimal Routing

False Resource Cycles can arise with minimal or nonminimal routing. Duato's incoherent routing algorithm is an example of the latter. We now present an example of a

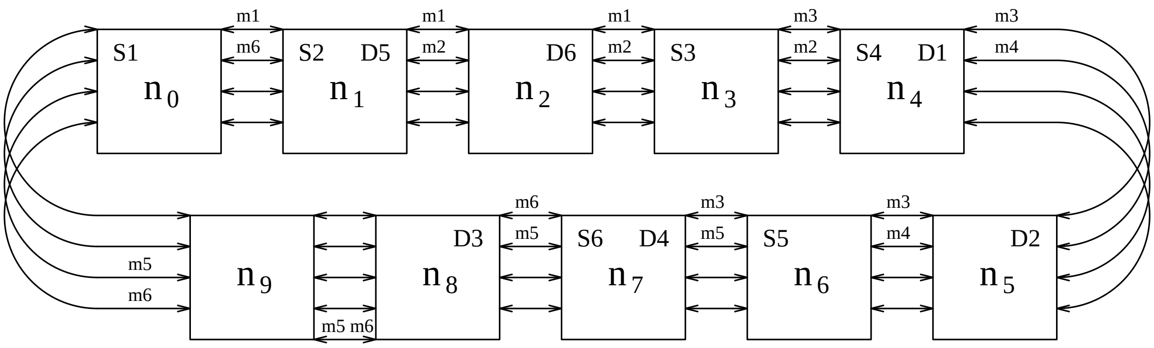


Figure 24: A False Resource Cycle for Minimal Routing

False Resource Cycle for minimal routing. The routing algorithm is presented for the ring (1D torus) shown in Figure 24, which also depicts a False Resource Cycle that occurs when the messages are routed in the clockwise direction. The multiprocessor shown in Figure 24 consists of ten nodes, denoted n_0 through n_9. There are four virtual channels per physical channel, except for the channel between nodes n_8 and n_9, which has five virtual channels. There are six messages active in the network, m_1 through m_6. In the upper left-hand corner of each node, Si is used to denote this node as the source of message m_i. Likewise, in the upper right-hand corner of each node, Di is used to denote this node as the destination of message m_i. Each message appears above each virtual channel that it currently occupies. For example, m_1 is using the first virtual channel between n_0 and n_1.

The routing algorithm permits a message to use VC_1 if the destination of the message is an even numbered node and VC_2 if the destination is an odd numbered node. A message on VC_1 or VC_2 stays on VC_1 or VC_2, respectively, until a wrap-around channel is used. The message then switches from VC_i to $VC_{(i+2) \bmod 4}$. In addition, any message routing from node n_8 to (or through) node n_9 is allowed to use VC_5. After a message uses VC_5, the message routes either on VC_3, if the destination is an odd numbered node, or on VC_4, if the destination is an even numbered node. It is possible to create a cycle in the CWG only if two messages arrive at node n_8 and *both* messages leave n_8 on VC_5. Clearly, both messages cannot acquire VC_5 at the same time, so this is a False Resource Cycle. There are no True Cycles in the CWG, so deadlock freedom follows immediately from theorem 4.2.

4.5.2 Testing for False Resource Cycles

In order to create a cycle, each message in the set, m_i, must acquire c_i' before m_{i-1} arrives at c_i'. This is always possible with a True Cycle, since each message occupies different channels. A False Resource Cycle, however, requires that at least two messages in the cycle share a channel. A False Resource Cycle can arise in two ways. Either a channel in the cycle is shared or a channel outside the cycle is shared by at least two messages in the cycle prior to entering the cycle. If the shared channel is part of the cycle, then each message, m_i, that uses the shared channel has already acquired c_i'. Hence, the False Resource Cycle would instead be a True Cycle every m_i could reach c_{i+1}' without sharing a channel. On the other hand, if the shared channel is used prior to the channels in the cycle, then a False Resource Cycle results from the inability of m_i to acquire c_i' before m_{i-1}. The False Resource Cycle would instead be a True Cycle if m_i could reach c_i' without using a shared channel.

The first possibility to consider is a False Resource Cycle involving only the channels that form the cycle. We need to consider channels at only those nodes that are used by more than one message in the cycle. A message that can form its part of the cycle without using any node used by another message in the cycle cannot be using a shared channel. Hence, ignore any message that can route through nodes that no other message in the cycle can use. Also, ignore any message that can route using channels that no other message in the cycle can use, even if other messages share a node with this message. For the remaining messages, see if a channel-disjoint path can be found for each message. This is done by arbitrarily choosing a path for one of these remaining messages. Continue selecting channel-disjoint paths for the other remaining messages. If a channel-disjoint path can be found for each message, then there is no False Resource Cycle formed with the channels used in the cycle. Otherwise, backtrack, selecting a different channel-disjoint path for the previous message. If all possible paths have been considered and there is still no channel-disjoint path for all messages, then this is a False Resource Cycle.

The second possibility to consider is a False Resource Cycle created only when at least two messages share a channel that is used prior to the cycle. This possibility is only considered if a channel-disjoint path within the cycle exists for each message in the cycle. Furthermore, this second possibility can occur only with routing algorithms that are not suffix-closed. If a routing algorithm is suffix-closed, then any cycle can be formed without using channels outside the cycle. For routing algorithms that are not suffix-closed, first ignore any message, m_i, that can route from c'_i to c'_{i+1} where the source of message m_i could be at c'_i. Next ignore any message that could route to c'_i using nodes that no other message in the cycle can use. Also, ignore any message that can route using channels that no other message in the cycle can use. For the remaining messages, consider the paths that a message could take to reach c'_i. Proceed as in the previous case, selecting channel-disjoint paths and backtracking as necessary. If a channel-disjoint path can be found for each message, then this is a True Cycle. Otherwise, this cycle may be a False Resource Cycle. We do not have an algorithmic method of determining this.

A message that uses a shared channel outside the cycle can be short enough to release this channel after it has been used and hold only the channels that form its part of the cycle. Hence, even though the channel is shared by more than one message in the cycle, the channel could be shared consecutively rather than simultaneously. The cycle is a True Cycle if the cycle can be formed when the shared channels are used consecutively rather than simultaneously. This is a False Resource Cycle, however, if even one shared channel must be used simultaneously by more than one message in the cycle.

4.6 Identifying CWG'

In this section, a formal design methodology for reducing the CWG to CWG' is described. This methodology is needed only for routing algorithms that do not require a message to wait for a specific output channel. The design methodology requires the identification of all cycles in the CWG. Since the number of cycles in the CWG could be exponential in the number of channels, finding a CWG' requires an exponential time algorithm. Other general techniques for proving deadlock freedom also require exponential time in the worst case [23, 27, 64]. The algorithm for reducing the CWG to CWG' has the following steps:

1. Create a list, L, of all cycles in the CWG. Each entry in L contains three sets: s_1, the set of edges that form the cycle, s_2, the set of edges in the cycle the algorithm has attempted to remove, and s_3, the set of edges that are currently removed from the cycle. Initially, $s_2 = s_3 = \emptyset$ for all cycles in L. Also keep

an ordered list of True Cycles that have been resolved. This list is used for backtracking.

2. Remove all False Resource Cycles from L. This is done by examining each cycle in L using the method described in Section 4.5 or informally by inspecting the CWG. Label the True Cycles TC_i $(i = 1, 2, \ldots, n)$ and let $i = 1$.

3. Remove an edge from TC_i as long as the routing algorithm remains wait-connected. Any edge that cannot be removed with the routing algorithm still wait-connected is added to s_2.

4. If every edge in TC_i is required for the routing algorithm to remain wait-connected, then backtrack to the previous True Cycle. Before backtracking, reset $s_2 = s_3 = \emptyset$ for TC_i. Remove the appropriate edge in s_3 from this previous True Cycle, but leave this edge in s_2, since it has already been tried.

5. Otherwise, TC_i can be resolved, so add this edge to s_2 and s_3. Choose the next True Cycle, from L with $s_3 = \emptyset$ and s_1 does not contain any edge that is in s_3 for a resolved True Cycle. Set i to the corresponding number for this cycle.

6. Repeat steps 3 – 5 until all True Cycles in L have been resolved or the routing algorithm has backtracked to TC_1 and the algorithm has already attempted to remove every edge TC_1. If all True Cycles in L have been resolved, then the routing algorithm is deadlock-free and CWG' consists of the edges that have not been removed. Otherwise, no acyclic CWG' exists for this routing algorithm, so the routing algorithm is not deadlock-free.

4.6.1 Example Reduction

Through inspection, we have proved deadlock freedom for the incoherent routing algorithm proposed by Duato. We now illustrate the formal methodology proposed in Section 4.6 by proving deadlock freedom for the same routing algorithm.

Initially, L is the list of all the cycles:

$$L = \left\{ \begin{array}{l} \{c_{A1}\}\{\emptyset\}\{\emptyset\}, \{c_{H1}\}\{\emptyset\}\{\emptyset\}, \{c_{B2}\}\{\emptyset\}\{\emptyset\}, \\ \{c_{A1}, c_{H1}\}\{\emptyset\}\{\emptyset\}, \{c_{A1}, c_{B2}\}\{\emptyset\}\{\emptyset\}, \{c_{H1}, c_{B2}\}\{\emptyset\}\{\emptyset\} \end{array} \right\} \tag{4.1}$$

Through either inspection or the application of the approach described in Section 4.5, cycle $\{c_{A1}, c_{H1}\}$ is shown to be a False Resource Cycle. Hence, this cycle is removed from L. All the other cycles are True Cycles. After labeling the cycles, L is:

$$L = \left\{ \begin{array}{l} TC_1 : \{c_{A1}\}\{\emptyset\}\{\emptyset\}, TC_2 : \{c_{H1}\}\{\emptyset\}\{\emptyset\}, TC_3 : \{c_{B2}\}\{\emptyset\}\{\emptyset\}, \\ TC_4 : \{c_{A1}, c_{B2}\}\{\emptyset\}\{\emptyset\}, TC_5 : \{c_{H1}, c_{B2}\}\{\emptyset\}\{\emptyset\} \end{array} \right\} \tag{4.2}$$

Set $i = 1$ and remove c_{A1} from TC_1. Since c_{H1} is also a waiting channel, the routing algorithm is still wait-connected. Likewise, set $i = 2$ and remove c_{H1} from TC_2 and then set $i = 3$ and remove c_{B2} from TC_3. Next, set $i = 4$ and remove c_{A1} from TC_4. The routing algorithm is still wait-connected, because the message that occupies c_{B2} can wait for c_{H1}. Finally, set $i = 5$ and remove c_{H1} from TC_5. The routing algorithm is no longer wait-connected, because a message that occupies c_{B2} cannot wait for c_{A1} or c_{H1}. Instead, remove c_{B2} from TC_5. This prevents a message on c_{H1} from waiting for c_{B2}, however, the routing algorithm is still wait-connected, because c_{H1} can wait for c_{H2}. Note that this is a different CWG' than one depicted in

Figure 23. All five True Cycles have been resolved and the routing algorithm is still wait-connected, so the routing algorithm is deadlock-free. L contains the following information:

$$L = \left\{ \begin{array}{l} TC_1 : \{c_{A1}\}\{c_{A1}\}\{c_{A1}\}, TC_2 : \{c_{H1}\}\{c_{H1}\}\{c_{H1}\}, \\ TC_3 : \{c_{B2}\}\{c_{B2}\}\{c_{B2}\}, TC_4 : \{c_{A1}, c_{B2}\}\{c_{A1}\}\{c_{A1}\}, \\ TC_5 : \{c_{H1}, c_{B2}\}\{c_{H1}, c_{B2}\}\{c_{B2}\} \end{array} \right\} \tag{4.3}$$

4.7 Routing Examples

In order to demonstrate the usefulness of the necessary and sufficient condition, a partially adaptive nonminimal routing algorithm for n-dimensional meshes is proposed. The routing algorithm does not require any virtual channels and does not have an acyclic channel dependency graph. Our methodology is also used to show deadlock freedom for a fully adaptive minimal hypercube routing algorithm. The necessary and sufficient condition is then used to prove that any relaxation of the restrictions imposed by the routing algorithm introduces the possibility of deadlock.

4.7.1 Mesh Routing Algorithm

We now propose a partially adaptive nonminimal routing algorithm for meshes that does not contain an acyclic channel dependency graph and does not require any virtual channels. (For a 2D mesh, this routing algorithm is similar to *north-last* proposed by Glass and Ni [39, 41], although our routing algorithm permits messages to make more 180° turns.) This new routing algorithm, called the Highest Positive Last routing algorithm, is defined below.

Let h be the highest dimension in which the message still needs to route in the *negative* direction to reach the destination. The following set of restrictions is applied to the channels:

- A message can use any channel in a dimension lower than h, even if the message does not need to route in that direction, so nonminimal routing is permitted. A message can also route in the negative direction of dimension h.

- A message that needs to route in only positive directions must use the positive channels in increasing dimension order, although the message is permitted to misroute in the negative direction of a higher dimension if desired.

- A message is allowed to make a 180° turn from the negative direction to the positive direction of the same dimension when the message needs to route in the positive direction. A message is permitted to make a 180° turn from the positive direction to the negative direction of the same dimension only when the message needs to route in the negative direction of this dimension and some higher dimension.

- If all output channels a message can use are busy, the message waits for the channel in the negative direction of dimension h. If the message needs to route only in positive directions, then the message waits for the channel in the positive direction of the lowest dimension in which the message needs to route.

The Highest Positive Last routing algorithm has a routing relation of the form $R : C \times N \times N \rightarrow C^p$. For example, consider a message that needs to route only

East and North. When the message is due South of its destination, the message is permitted to route South if the input channel was East, however, the message cannot route South if the input channel was North. (If this restriction is not imposed, the routing algorithm is not deadlock-free.) The routing algorithm is not coherent even for minimal paths. For example, on an $8 \times 8 \times 8$ mesh, a message from $(4,4,2)$ to $(5,5,1)$ can route through $(4,5,2)$, however, a message from $(4,4,2)$ to $(5,5,2)$ cannot route through $(4,5,2)$. Finally, the routing algorithm does not have an acyclic channel dependency graph, but does have an acyclic channel waiting graph. For example, a message that needs to route in the negative direction of the Z dimension can use the channels in the X and Y dimensions in any order, but always waits for the channel in the negative direction of the Z dimension. Duato's proof technique cannot be applied to the Highest Positive Last routing algorithm, because the routing relation is of the form $R : C \times N \times N \rightarrow C^p$.

The routing algorithm allows substantially more minimal paths than any previously proposed mesh routing algorithm that requires only one channel. Previous partially adaptive routing algorithms have required an acyclic channel dependency graph. For example, Glass and Ni [41] propose the *negative-first* routing algorithm for n-dimensional meshes. The negative-first routing algorithm requires messages to route in all negative directions and then all positive directions. The negative-first routing algorithm also permits nonminimal routing on the negative channels before using any positive channels. Glass and Ni prove that an acyclic channel dependency graph prohibits at least $n(n-1)$ 90° turns for an n-dimensional mesh. With negative-first routing, a message cannot use a negative channel after using a positive channel, so each of the n positive channels is prohibited from using the negative channel in the other $n-1$ dimensions. The Highest Positive Last routing algorithm does not allow a message to use a channel in a lower dimension after using the positive channel in a higher dimension, which restricts $n(n-1)$ 90° turns. However, the restrictions imposed by the Highest Positive Last routing algorithm are not absolute. A message can use a positive channel before a channel in a lower dimension whenever the message needs to route in the negative direction of some still higher dimension. This is a significant relaxation of the routing restrictions, especially on the positive channel in the lower dimensions. Although the positive channel in dimension i cannot be used before the positive or negative channel in a lower dimension, this restriction is removed when the message also needs to route in the negative direction of any dimension higher than i. The lower the dimension, the more likely this restriction can be relaxed.

Theorem 4.4 *The Highest Positive Last routing algorithm is deadlock-free.*

Proof. This proof uses the necessary and sufficient condition proved in theorem 4.2. Any cycle in the CWG requires at least two dimensions, since a message routing in the positive direction cannot *wait for* a channel in the negative direction without first routing in the negative direction of another dimension. Consider any potential cycle in the CWG involving the channels used by the messages that form the cycle. It is possible to partition this cycle into two hyperplanes by dividing the cycle along the highest dimension used in the cycle. For the cycle to exist, some message must use a channel in the positive direction of the highest dimension in the cycle and then wait for a channel in the positive half of the cycle. In other words, some message must cross the partition in the positive direction and then wait for a channel in this half of the partition for the cycle to exist. A message is not permitted to use a channel in the positive direction and then wait for a channel in a lower dimension without first using a channel in the negative direction of a higher dimension. Since the message has already used the positive direction of the highest dimension in the cycle, the message has not used the negative channel in a still higher dimension. Thus, no message in the cycle can satisfy this requirement and the CWG is acyclic. Deadlock freedom follows immediately from theorem 4.2. □

4.7.2 Hypercube Routing Algorithm

The following conventions are used in this section. Channels are assumed to be bidirectional virtual channels. $VC^i_{n\pm}$ is used to denote virtual channel n in the $\pm$ direction of dimension i. For example, VC^3_{1-} is virtual channel one in the negative direction of the third dimension. An asterisk in the superscript denotes all dimensions. Thus, VC^*_{2-} denotes the second virtual channel in the negative direction of all dimensions. A message from S to D routes in the *positive* direction of dimension i if $s_i = 0$ and $d_i = 1$. Similarly, the message routes in the *negative* direction of dimension i if $s_i = 1$ and $d_i = 0$. With minimal routing, a message routes in each dimension at most once and a message does not route in dimension i if $s_i = d_i$.

We now propose a fully adaptive minimal routing algorithm for hypercubes. All previous fully adaptive routing algorithms for hypercubes require nonadaptive routing on the first set of virtual channels. This new routing algorithm permits partially adaptive routing on the first set of virtual channels. The routing algorithm, called the Enhanced Fully Adaptive routing algorithm, is defined by the following steps:

- Assign two bidirectional virtual channels to each dimension.
- Allow a message to route along the second virtual channel at any time.

Let l be the lowest dimension in which the message still needs to route. The following set of restrictions is applied to the first set of virtual channels:

- A message that needs to route in the negative direction of dimension l can use any of the first set of virtual channels.
- A message that needs to route in the positive direction of dimension l *must* use VC^l_{1+}.
- If all output channels a message can use are busy, the message waits for VC^l_1.

Note: Alternatively, this routing algorithm could allow a message to wait for *any* output channel it is permitted to use. For this alternative, CWG' is restricted to the first virtual channel in the lowest dimension in which the message needs to route. CWG' is then equivalent to the CWG just defined. Hence, the proof of deadlock freedom is unchanged, except that theorem 4.3 is used to prove deadlock freedom.

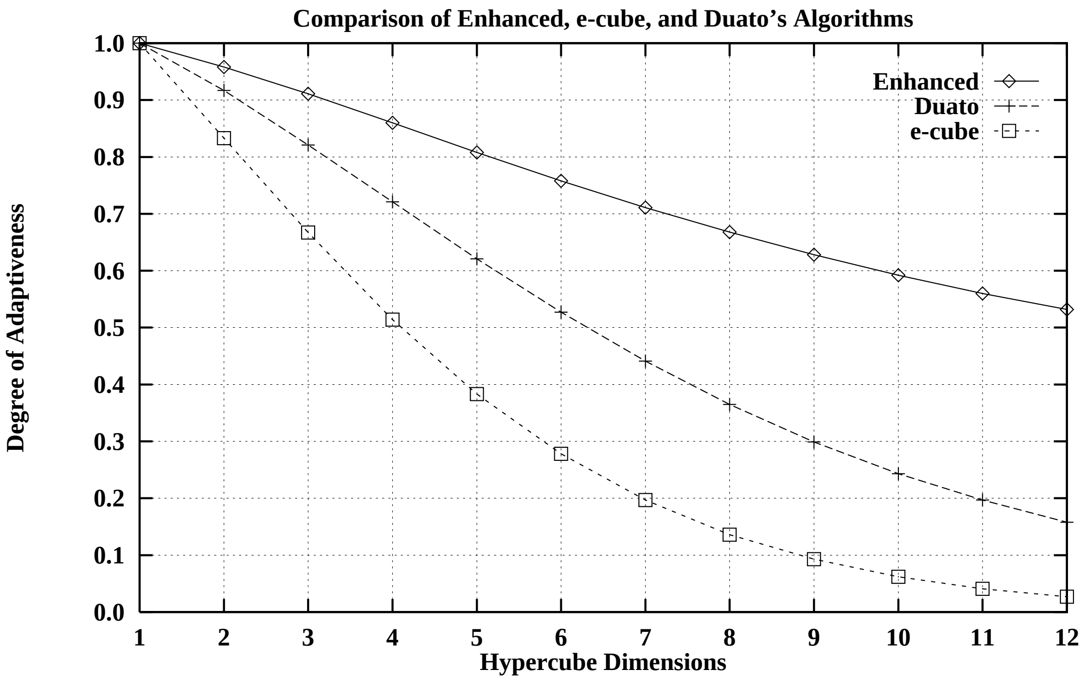


Figure 25: Degree of Adaptiveness for Hypercube Routing Algorithms

This routing algorithm is substantially more adaptive than any previously proposed fully adaptive hypercube routing algorithm. The Enhanced Fully Adaptive routing algorithm restricts a message from using the first virtual channel in the positive direction after using the first virtual channel in a higher dimension, however, a message can use the first virtual channel in a higher dimension first whenever the message needs to route in the negative direction of the lowest dimension in which the message still needs to route. This is a significant relaxation of the routing restrictions, especially compared with using dimension-order routing for the first set of virtual channels.

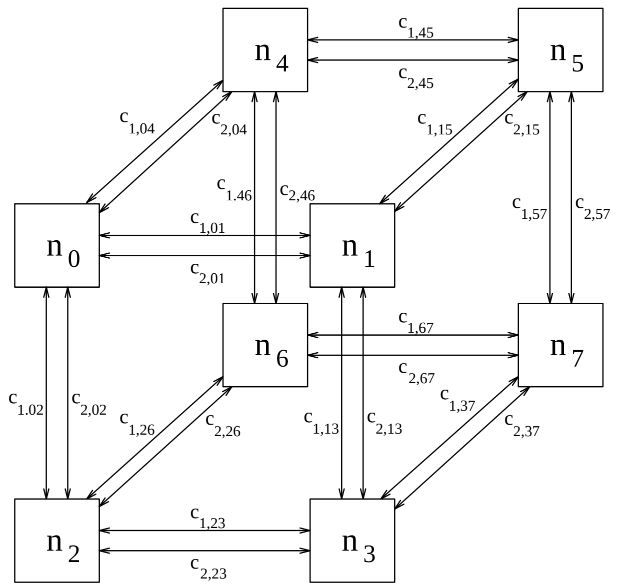


Figure 26: 3D Hypercube with 2 Virtual Channels per Physical Channel

To illustrate the increase in adaptiveness, we present the degree of adaptiveness of both routing algorithms in Figure 25. For comparison, the degree of adaptiveness for *e*-cube (nonadaptive dimension-order) routing is also shown. Paradoxically, the degree of adaptiveness is not zero for nonadaptive routing, because nonadaptive routing always allows one of paths and the degree of adaptiveness would be zero only if there were *no* permitted paths. There are relatively few paths when the source is near the destination, so even a single path can represent a relatively large proportion of the paths. For example, nonadaptive routing can use half the paths when the distance between the source and destination is two hops.

Figure 26 depicts a 3D hypercube and Figure 27 depicts the acyclic CWG for the first set of virtual channels on a 3D hypercube. The second set of virtual channels is not shown. Since no message waits for a second virtual channel, there are no edges in the CWG *to* a second virtual channel. For this reason, any cycle in the CWG must be created among the first virtual channels. Note that although the second virtual channels are not shown in Figure 27, dependencies that arise among the first set of virtual channels through the use of the second set of virtual channels are shown.

Theorem 4.5 *The Enhanced Fully Adaptive routing algorithm is deadlock-free.*

Proof. This proof uses the necessary and sufficient condition proved in theorem 4.2. R_A is wait-connected, since a message is always permitted to wait for virtual channel one in the lowest dimension in which the message needs to route. A message can wait for only VC_1^*, so any cycle in the CWG must be created from waiting dependencies among the VC_1^* channels (although VC_2^* channels could be used to create these dependencies). Since minimal routing is used, any cycle in the CWG requires at least two dimensions and must use both directions of each dimension in the cycle. Let l be the lowest dimension of a potential cycle. A cycle in the CWG requires one of two situations: either a message, m_i, waits for VC_{1+}^l after using VC_1^* in a higher dimension of the cycle or m_i uses VC_{1+}^l or VC_{2+}^l after using VC_1^* in a higher dimension and then waits for VC_1^* in a different higher dimension. (Since l is the lowest dimension in the cycle, m_i does not need to route in any dimension lower than l after using VC_1^* in a higher dimension. Otherwise m_i would be waiting for a channel in this lower dimension and l would not be the lowest dimension in the cycle.) Clearly, neither situation can occur, since m_i cannot use VC_1^* in a higher dimension

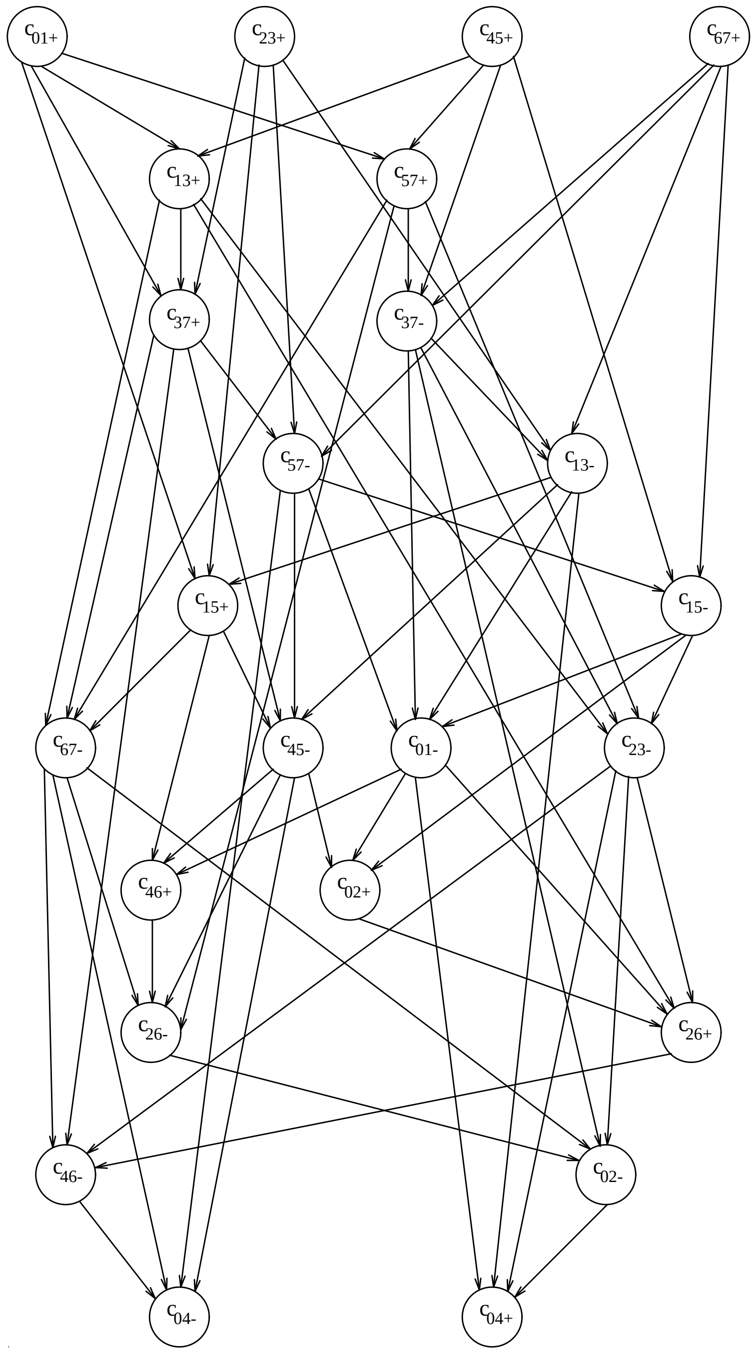


Figure 27: The CWG for the Enhanced Fully Adaptive Routing Algorithm

when m_i needs to route in VC_+^l and l is the lowest dimension in which m_i needs to route. Therefore, the routing algorithm is deadlock-free. □

The Enhanced Fully Adaptive routing algorithm is not coherent. For example, a message from n_5 to n_2 can route through n_1 using the first virtual channel and then route through n_3, however, a message from n_5 to n_3 cannot route through n_1 using the first virtual channel. Thus, the Enhanced Fully Adaptive routing algorithm is not coherent, because it is not prefix-closed. Since the routing algorithm is not coherent, Duato's proof technique cannot be used to prove theorem 4.6.

Theorem 4.6 *No restrictions imposed on the Enhanced Fully Adaptive routing algorithm can be relaxed without permitting a deadlock configuration.*

Proof. There are no restrictions on the use of the second set of virtual channels, so the restrictions on only the first set of virtual channels must be considered. The only restriction on the first set of virtual channels is that a channel in a higher dimension cannot be used by a message that needs to route in the positive direction of the lowest dimension in which the message needs to route. Without loss of generality, let l be the lowest dimension and assume that the channel used in a higher dimension is the first virtual channel in the positive direction of dimension i. There is now an edge in the CWG from VC_{1+}^i to VC_{1+}^l. The CWG already has edges from VC_{1+}^l to VC_{1-}^i, from VC_{1-}^i to VC_{1-}^l, and from VC_{1-}^l to VC_{1+}^i. These edges form a True Cycle. A message waits for only VC_1^l and the routing algorithm is not deadlock-free if VC_1^l is not a waiting channel. Thus, from theorem 4.2, the routing algorithm is no longer deadlock-free. □

4.8 Universality of Our Methodology

The proposed necessary and sufficient condition is not specific to wormhole routing. In fact, this proof technique can be applied to any switching technique that relies on routing restrictions to guarantee deadlock freedom. Hence, our methodology is a *universal* proof technique. Other methods of avoiding deadlock, such as abort/retry, which is used in circuit switching, and misrouting, which is used in adaptive cut-through, are inherently deadlock-free.

In this section, we focus on extending the necessary and sufficient condition to store-and-forward routing and virtual cut-through. Unlike wormhole routing, store-and-forward routing and virtual cut-through use central buffers, called standard buffers, to store packets. (Alternatively, edge buffers can be associated with each physical channel. Edge buffers are different from a logical standpoint, but, like central buffers, are physically located in system memory on the processor.) Injection and delivery buffers are (logically) used to inject and deliver packets, respectively. Virtual cut-through and store-and-forward routing require that each buffer is large enough to store an entire packet. Messages of arbitrary length can be generated, but each packet has a maximum size. (Without loss of generality, we will assume that each buffer can hold exactly one packet.)

Virtual cut-through allows packets to span several nodes. Once a packet is blocked, however, that packet is buffered at the node where it was blocked. As previously noted, a deadlock configuration consists of a set of waiting packets that are permanently blocking each other. Hence, any deadlock configuration with virtual cut-through is indistinguishable from the corresponding deadlock configuration with store-and-forward routing. That is, every packet occupies exactly one buffer and is waiting for a buffer occupied by another buffer in the set.

Buffers are being used instead of virtual channels. (Technically speaking, wormhole routing uses buffers, the buffer associated with the corresponding virtual channel.) For this reason, the term *buffer waiting graph* (BWG) is used in place of channel waiting graph. Note, however, that the definition is unchanged. In other words, there is an edge in the BWG from buffer b_i to buffer b_j if a packet that occupies buffer b_i can wait for buffer b_j. Given the conditions for deadlock with store-and-forward routing and virtual cut-through, these dependencies must be between buffers either on the same node or on neighboring nodes. In other words, a dependency in the BWG from b_i to b_j exists only if a packet in b_i waits for buffer b_j *immediately* after using buffer b_i.

For store-and-forward routing, routing algorithms are defined with routing relations of the form $R : B \times N \times N \rightarrow B^p$ or $R : N \times N \times N \rightarrow B^p$. That is, the routing decision is based on the current buffer and source and destination nodes or the current node and the source and destination nodes. As mentioned before, more general definitions of the routing relation can be supported. The idea of waiting channels is independent of the routing relation and selection function, so our proof technique is valid for other definitions of the routing relation and selection function.

The necessary and sufficient condition proposed for wormhole routing in theorem 4.2 and theorem 4.3 also holds for store-and-forward routing and virtual cut-through. These theorems are defined based on the relationship between channels occupied by a packet and the channel(s) for which a packet is waiting. For store-and-forward routing and virtual cut-through, the relationship is between the packet buffers. Specifically, True Cycles (and hence deadlock configurations) are formed from a set of packets waiting for buffers occupied by other packets in the cycle. The BWG is still based on the relationship between occupied buffers and waiting buffers with store-and-forward routing and virtual cut-through. Hence, deadlock configurations require True Cycles. Of course, all edges in the BWG are between buffers on the same node or neighboring nodes. This restriction of edges in the BWG to only neighboring buffers should actually simplify the *application* of these theorems, especially for identifying an appropriate BWG'.

We believe that just as we have identifed BWG, a similar "waiting graph" can be identified for any other proposed switching technique that relies on routing restrictions to achieve deadlock freedom.

To illustrate the simplicity of applying our methodology to store-and-forward routing algorithms, we present a fully adaptive minimal n-dimensional mesh routing algorithm that is more adaptive than the routing algorithm proposed in [83]. This example demonstrates that reducing the BWG to BWG' is not difficult – we suspect that this will be the case with many other routing algorithms.

This routing algorithm, called the *Two-Buffer* routing algorithm, requires two standard buffers per node, labeled A and B. A packet uses only the A buffers until the packet has completed routing in the positive direction of *all* dimensions. Since minimal routing is used, the packet can move to the A buffer of *any* neighboring node that moves the packet closer to its destination. Once the packet has completed routing in all positive directions, the packet remains in the A buffers as long as at least one of its output buffers is available, however, if the message is blocked, it moves to the B buffers and routes in the B buffers until the destination is reached.

Theorem 4.7 *The Two-Buffer routing algorithm is deadlock-free.*

Proof. The B buffers are used only when routing in the negative directions and a packet can never move from a B buffer to an A buffer. Since there are no wrap-around channels on a mesh, a cycle cannot be created using the B buffers. Now consider the A buffers. There are no restrictions on the use of the A buffers. Hence, there are cycles in the BWG. Define BWG' such that all edges in the BWG from an A buffer to an A buffer in the negative direction are removed. Note that BWG' remains wait-connected, since a blocked packet remains in the A buffers only if it needs to route in the positive direction of at least one dimension. Also, BWG' is acyclic, since there are no wrap-around channels and all the edges are either from an A buffer to another A buffer in a positive direction or from an A buffer to a B buffer or from a B buffer to another B buffer in a negative direction. Deadlock freedom follows immediately from theorem 4.3. $\square$

4.9 Summary

In this chapter, we have presented a necessary and sufficient condition for deadlock-free routing. This necessary and sufficient condition provides a unifying framework for proving deadlock freedom for any switching technique that relies on routing relations to achieve deadlock freedom. Our methodology has been applied to routing algorithms for meshes and hypercubes, for nonminimal and minimal routing, and for wormhole routing, store-and-forward routing and virtual cut-through. This concludes the presentation of our contributions to adaptive routing. In the next chapter, we consider the mapping problem; our interest is in producing mappings that reduce channel contention.

CHAPTER V

Contention-Sensitive Mapping Heuristic

So far, we have presented results for only adaptive routing, which is an important architectural approach to reducing channel contention. As discussed in Chapter I, mapping can also be used to reduce contention. Techniques for generating efficient mappings of parallel algorithms are necessary to ensure effective utilization of parallel architectures. Reducing communication overhead has been widely recognized as a requirement to achieving these efficient mappings. This chapter presents an iterative heuristic for compile-time mapping of clustered task graphs. Special attention is given to measuring and reducing channel contention. An example of the different ways in which channel contention can arise is presented in Section 5.1. Our mapping heuristic is then described in Section 5.2. Experimental results are presented in Section 5.3 to show the effects of channel contention for packet-switched networks and the improvement realized by our heuristic. A published version of the results in this chapter can be found in [89].

5.1 Examples of Channel Contention

Figure 28 illustrates the different ways that channel contention can result in message delay. In Figure 28, assume that the clusters have been mapped onto a 2D hypercube.

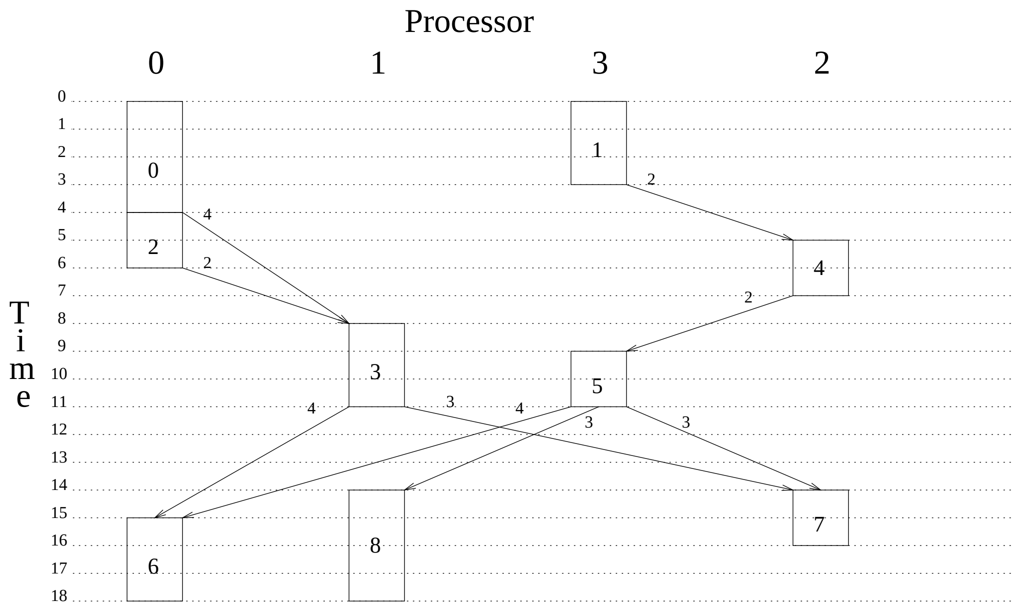


Figure 28: Examples of Contention-Induced Delays

All tasks in a column are in the same cluster. The horizontal axis identifies the processor for the corresponding cluster. The vertical axis shows the time. Each rectangle represents a task with the associated task number inside the rectangle. The execution time of a task is the length of its rectangle. Each directed edge is a message. The weight of each edge indicates the minimum time required to transmit that message.

There are three different situations in which channel contention potentially increases the transmission times of messages:

- Finite bandwidth on a channel. Consider the message from task 0 to task 3 and the message from task 2 to task 3. Both of these require the same channel, however, task 0 is still using this channel when task 2 finishes. For this reason, it is necessary to delay the message from task 2 to task 3.

- A limited number of channels at each processor. There are three messages being transmitted by task 5. Since this processor has only two channels, one of the messages must be delayed.

- Limited paths through the interconnection network. Task 3 is sending its messages at the same time that task 5 is sending its messages. The message from task 3 to task 6 can begin transmission immediately. The message from task 3 to task 7 must either wait until the message from task 3 to task 6 has completed transmission or be routed through the processor that executed task 5. Since task 5 is already using both of its channels, the message must be delayed in either case.

Since the mapping algorithm proposed by Yang, Bic, and Nicolau [102] and our mapping algorithm assume clustering as a prior step, the two heuristics are compared through simulation. Our mapping heuristic has at least two significant differences. It allows the reassignment of critical clusters and it considers the channel contention in each mapping. Simulations show the benefit of these differences. The mapping heuristic in [102] assumes a total ordering of the tasks in a cluster. For comparison purposes, we make the same assumption, however, our mapping heuristic does not require a total ordering of the tasks. For example, the DAGs used in Section VI do not impose a total ordering on the tasks in a cluster.

5.2 The Mapping Heuristic

5.2.1 Overview

Recall that there are p processors in the parallel architecture and there are V tasks in the parallel algorithm. In general, $V \gg p$. There are a total of E messages in the parallel algorithm. The algorithm uses a function, *distance*(), which returns the length of the shortest path between any two processors. When analyzing the time complexity of the mapping algorithm, we assume that this function can be computed in constant time.

Our heuristic uses the critical path to attempt to iteratively improve the mapping. *Any edge that increases the total execution time if its transmission is delayed is a critical edge.* Any task that sends or receives a critical edge is a critical task. Any cluster that contains at least one critical task is a critical cluster. Each task is assigned an execution window. This is the time range in which a task must begin execution in order to avoid increasing the execution time of the task graph.

In Figure 29, assume that the clusters have been mapped onto a 2D hypercube. The axes, rectangles, and directed edges are defined as in Figure 28. We refer to Figure 29 throughout the presentation of the mapping heuristic.

5.2.2 Calculate the Communication Matrix

Since the amount of communication among clusters is independent of the mapping, this step is performed once and the results are reused for each iteration. Each message has a sending task and a receiving task that are assigned to different clusters.

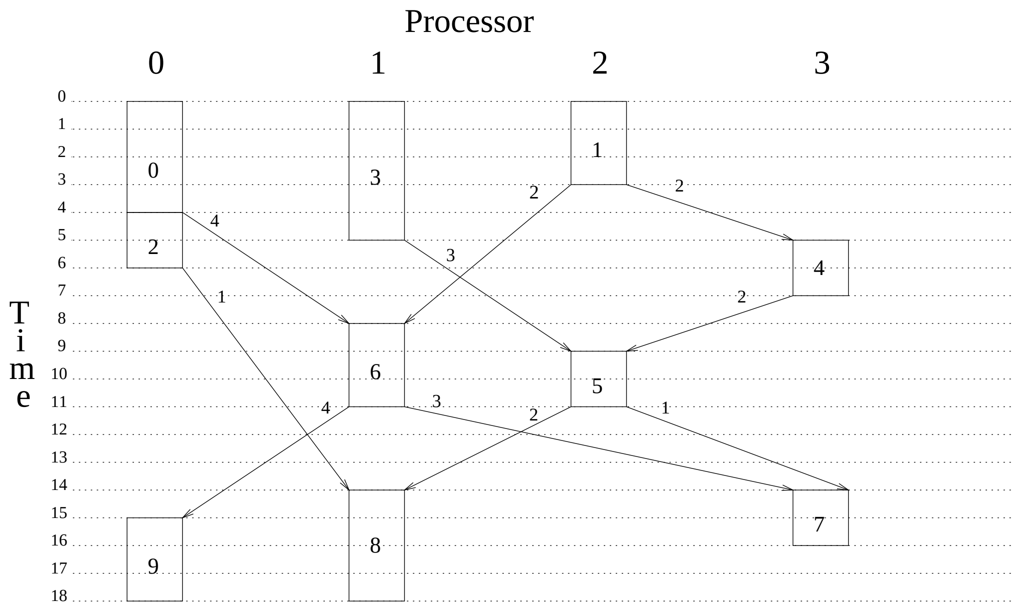


Figure 29: A Sample Task Graph

The amount of communication between different clusters is stored in a $p \times p$ matrix. We add the length of each message to the corresponding elements of the communication matrix. The direction of the message is ignored. After all the messages have been processed, each element in the matrix corresponds to the total message traffic between the two specified clusters. The amount of communication involving each cluster is also totaled. This information is used when mapping the clusters that are not critical. Since all edges must be examined once and all elements of the matrix must be initialized to zero, the time complexity of this step is $O(E + p^2)$.

5.2.3 Mapping the Clusters to the Processors

A mapping is first generated on a completely connected architecture. The execution time of this mapping is used as a lower bound for an optimal mapping on the target multiprocessor. Information from the mapping on the completely connected architecture is also used as input to the first iteration of our heuristic. The remaining iterations use the results from the previous iteration.

Create a Mapping for the Completely Connected Architecture

Map the clusters onto the completely connected architecture. This is done using the identity mapping. Hence, the time complexity of this step is $O(p)$.

Create a Mapping of the Critical Clusters

A mapping that reduces the transmission time of the critical edges improves the total execution time, because these edges have a direct impact on the total execution time. The heuristic used for mapping these critical clusters is to minimize the total distance between the source and destination of the critical edges. Since the critical edges directly affect the total execution time, this increases the probability that the mapping has a shorter total execution time.

The total distance of the critical edges is calculated for the current mapping. The algorithm then swaps each critical cluster with every other cluster one at a time and recalculates the total distance. The two clusters are swapped back only if the total distance has increased. The total distance for the initial assignment can be computed in $O(p^2)$ time. In the worst case, all the clusters are critical clusters, so $O(p^2)$ different swaps are tried. Since only two clusters are swapped each time, the total distance for each new assignment can be computed in $O(p)$ time. Thus, the time complexity of this step is $O(p^3)$.

Create a Mapping of the Other Clusters

Usually some clusters are not critical clusters. These clusters must also be mapped to minimize the overhead of transmitting their messages. The heuristic attempts to minimize the transmission time of the messages in these clusters by mapping these clusters next to clusters that share a lot of communication. This is of secondary importance to the critical edge assignment, so these clusters are assigned only to available processors. Any potential channel contention is ignored during the assignment. The heuristic attempts to minimize the channel contention experienced by these messages by shortening the distance traveled in the interconnection network.

The non-critical clusters are assigned by repeatedly selecting the available processor neighboring an assigned cluster with the most communication. The unassigned cluster with the most communication with the clusters on neighboring nodes of this processor is assigned to this processor. The goal is to maximize the number of messages requiring only one channel. At most, $O(p)$ clusters must be assigned using this method, selecting from at most $O(p)$ available processors. Checking the amount of communication with the neighboring nodes requires $O(\log p)$ time on a hypercube, so the time complexity of this step is $O(p^2 \log p)$.

5.2.4 Compute the Earliest Starting Times

In order to determine the execution window of each task, it is necessary to compute the earliest starting time of each task. Any delay due to channel contention is ignored in this step, but the delay due to distance is included. The channel contention is determined in Step 5.2.6.

In this algorithm, we make use of the assumption that the tasks in a cluster have a total ordering. This enables the algorithm to process the tasks in increasing order. The earliest starting time is the *latest* of the following two conditions:

- All the messages for this task have arrived. For example, task 6 in Figure 29 cannot start until time 8, because it must wait for the message from task 0.
- The previous task on this processor has completed execution. This is time at 0 for the first task in each cluster. For example, task 2 in Figure 29 cannot start until time 4.

The earliest that task 8 in Figure 29 can start is time 14. Even though the message from task 5 to task 8 has a length of two, $distance(5, 8) = 2$ (because a hypercube with the identity mapping is being used), so the message requires at least three time units to be transmitted.

When each task is evaluated, the messages that it receives are from tasks whose earliest starting time has already been computed. The earliest starting time of the previous task in this cluster is also known. This allows the algorithm to determine the earliest starting time of this task immediately. This must be done once for each task and requires that each communication message be processed once. Therefore, the time complexity of this step is $O(V + E)$.

5.2.5 Compute the Latest Starting Times

To determine the execution window of each task, the latest starting time of each task must also be computed. Since we are interested in achieving the minimum total execution time, the latest starting time chosen must be early enough that it does not increase the total execution time of the algorithm. Again, any delay due to channel contention is ignored in this step, but the delay due to distance is included. The channel contention is determined in Step 5.2.6.

We again use the assumption that the tasks in a cluster have a total ordering. In this case, however, it is necessary to process the tasks in decreasing order. The latest starting time is the *earliest* of the following two conditions:

- Start early enough that the task completes execution in time for the next task on this processor to also meet its deadline. This is the latest starting time of the next task minus the execution time of the current task. For the last task on each processor, this is the total execution time minus the execution time for this task. For example, the latest starting time for task 7 in Figure 29 is at time 16.

- Start early enough that all the messages the task generates arrive by the end of the execution window of the destination tasks. For example, the latest starting time for task 2 in Figure 29 is at time 11.

The latest that task 3 in Figure 29 can start is time 0. Even though the message from task 3 to task 5 has a length of three, $distance(3,5) = 2$, so the message requires at least four time units to be transmitted.

When each task is evaluated, the messages that it sends are to tasks whose latest starting time has already been computed. The latest starting time of the next task in this cluster is also known. This allows the algorithm to determine the latest starting time of this task immediately. This must be done once for each task and requires that each message be processed once. Therefore, the time complexity of this step is also $O(V + E)$.

5.2.6 Evaluate the Channel Contention

Figure 28 illustrates the instances where there could be delay due to channel contention. It it important to measure the delay experienced due to the channel contention. This results in a better estimate of the total execution time for the mapping. Since the total execution time of the mapping is more accurately determined, inferior mappings can be correctly rejected. In order to simulate the message transmission, the interconnection network and its properties must be defined. Fully adaptive minimal store-and-forward routing is assumed in this chapter.

Our measurement of the channel contention is much more accurate than the method proposed in [30]. We also measure the actual contention, rather than trying to minimize it using explicit control over the routing of messages. This allows the channel contention to be approximated based on the routing algorithm. By not imposing an arbitrary fixed path on messages, the routing of all messages is deadlock-free if the routing algorithm is deadlock-free.

In order to evaluate the impact of channel contention on the mapping, it is necessary to consider the traffic on the interconnection network during the entire execution of the algorithm. This execution is modeled with an event-driven simulator. An event occurs when a task completes execution (and places messages onto the outgoing channel queues), when a packet enters the network (and acquires channels), and when a packet leaves the network (and releases channels).

The time complexity of this step is partially dependent on the diameter of the parallel architecture, because the diameter of the interconnection network depends on the topology. For complexity analysis, it is assumed that the diameter is $\log p$, the actual value for a hypercube. Each task generates only one event. If the maximum length of any message is fixed, then the time complexity of this step is $O(V + E \log p)$. If a 2D mesh is used, the diameter is $\sqrt{p}$, so the time complexity increases to $O(V + Ep^{1/2})$.

5.2.7 Find the Critical Edges

The critical edges are used to guide the search for an optimal mapping on the target architecture. Since these edges cannot be delayed without increasing the total execution time, preference needs to be given to these messages when the clusters are being assigned. If it is possible to assign the clusters so that each of these messages is transmitted between neighboring processors in the target architecture, then there is a better chance of producing an optimal mapping. In order to determine the critical edges, it is only necessary to consider edges between tasks that have an execution window with no flexibility. In other words, the earliest starting time and the latest starting time are the same. All edges must be examined at most once, and all elements of the matrix must be initialized to zero, so the time complexity of this step is $O(E+p^2)$. Next, we prove that this approach correctly determines the set of critical edges.

Theorem 5.1 *All critical edges are transmitted between two tasks that have an execution window with no flexibility.*

Proof. Assume that a directed edge between tasks i and j, e_{ij}, is a critical edge. Without loss of generality, assume that task i is the source of the edge and task j is the destination. By definition, any delay in sending or receiving e_{ij} increases the total execution time. In order to minimize the transmission time of e_{ij}, task i must send the message as soon as possible. Since any delay increases the total execution time, $earliest(i) = latest(i)$. Similarly, task j must start as soon as the message is received, so $earliest(j) = latest(j)$. Therefore, both task i and task j must have an execution window with no flexibility. □

The converse is not necessarily true. For example, in Figure 29, the message from task 1 to task 6 is transmitted between two tasks that have an execution window with no flexibility. However, this edge is not critical. An edge is critical only if both execution windows have no flexibility and the transmission time of the message is equal to the difference between the ending time of the sending task and the starting time of the receiving task.

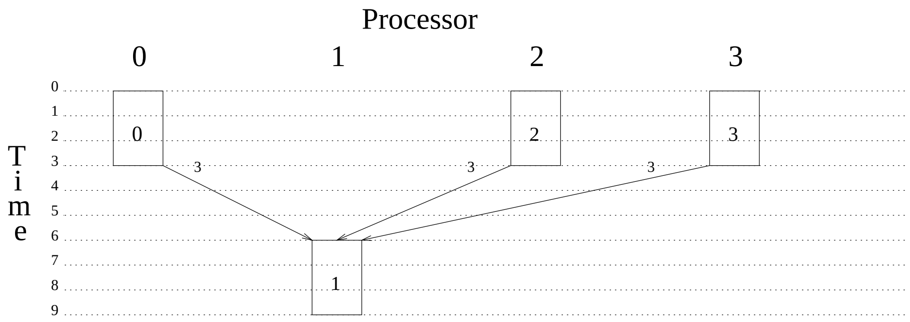


Figure 30: The Set of Critical Edges is Dependent on the Mapping

The set of critical edges in Figure 29 is $\{(0,6),(1,4),(3,5),(4,5),(5,8),(6,9)\}$. Accordingly, the execution windows for only tasks 2 and 7 in Figure 29 have any flexibility.

The set of critical edges must be recomputed for each iteration of the algorithm because they can change depending on the mapping. This is ignored in [102], which reduces the ability of their algorithm to find a satisfactory mapping. Since the mapping heuristic uses the set of critical edges to choose a mapping, this set must be recomputed after each iteration. Figure 30 presents a trivial example that illustrates the relationship between the mapping and the set of critical edges.

Assume the clusters have been mapped onto a 2D hypercube. The axes, rectangles, and directed edges are defined as in Figure 28. If the identity mapping is used, then the critical edge is the message from task 2 to task 1. If clusters 2 and 3 are swapped, then the message from task 2 to task 1 is no longer a critical edge. For this mapping, the critical edge is from task 3 to task 1.

5.2.8 Calculate the Total Execution Time

The goal of the mapping heuristic is to minimize the total execution time of the parallel algorithm. We now prove that, within our model, a mapping with a total execution time less than that of the completely connected architecture is impossible, so this time is a lower bound on the total execution time. This proof holds only for nonadaptive routing. It is unlikely that a mapping on the target architecture can achieve this lower bound even with adaptive routing. Hence, this time is used as the lower bound for all the mappings.

Theorem 5.2 *For a given clustering, no mapping on the target architecture can have a total execution time that is less than the total execution time of the identity mapping on the completely connected architecture.*

Proof. With nonadaptive routing, every message has only one path from the source to the destination. For the completely connected architecture, this path uses only a single channel. For the target architecture, this path requires at least one channel. Furthermore, the channels are shared among the processors, so there is the potential for additional channel contention. Any channel contention increases the transmission time. Hence, no message can arrive sooner on the target architecture than on a completely connected architecture. Thus, a mapping never executes faster on the target architecture than on the completely connected architecture. □

The total execution time is the time when the last executing task finishes. We check the ending time of each task, so the time complexity of this step is $O(V)$.

Note that this time may not be the optimal execution time. It is possible that no mapping can achieve this lower bound due to communication delays on the target architecture, however, finding the optimal execution time is equivalent to finding the optimal mapping. Since this problem is NP-hard, we use this estimate of the lower bound rather than generate the actual optimal execution time.

5.2.9 Termination

It is necessary to balance the complexity of the algorithm with the need to generate an efficient mapping. In order to maintain a balance between these two goals, three termination conditions are utilized:

- A previous assignment of the clusters to the processors has been repeated. Since the assignment process is deterministic, a cycle of assignments has been found. Continued execution will only repeat previous assignments. The time complexity of checking this condition is $O(p^2)$.

- A mapping with an execution time equivalent to that of the completely connected architecture has been achieved. Since a mapping superior to this one is very unlikely, the heuristic terminates.

- In order to guarantee an upper bound on the time complexity of the mapping algorithm, it is necessary to provide an additional termination condition. We terminate the mapping algorithm after p iterations. Based on measurements taken while computing the results presented in Section 5.3, this seems to give a reasonable trade-off between complexity and performance. Using p iterations was noticeably better than using roughly $p^{1/2}$ iterations and only slightly worse than using p^2 iterations.

5.2.10 Complexity

There are p iterations of the algorithm in the worst case. Examining each step of the algorithm shows a worst-case time complexity of $O(p^4 + Vp + Ep\log p)$. Since most of the steps have a small constant (in some cases less than one), this provides a good indication of the running time of the algorithm.

5.3 Simulation Results

All tests assume packet switching on a hypercube. All task graphs are randomly generated using task computation times that randomly vary from $1 - 10$. Similarly, message lengths randomly vary from $1-10$ packets and each task generates from $1-5$ messages. The tasks are uniformly distributed among the clusters. These simulation results are used to compare the five different mapping heuristics described below:

- **Ours No B/W** Our mapping heuristic is changed to ignore channel contention. The best mapping found is then simulated to determine the channel contention and to provide a more realistic assessment of the performance of the chosen mapping. This corrected measure of the total execution time is the time that appears in the results. The reevaluation of the chosen mapping is done only after the mapping algorithm has terminated and only for the sake of comparison.

- **Ours** Our mapping heuristic is used to choose a mapping.

- **YBN** The mapping heuristic proposed by Yang, Bic, and Nicolau [102] is used to choose a mapping. The critical clusters are not reassigned after their initial assignment, and channel contention is ignored. The best mapping found is then simulated to determine the channel contention and to provide a more realistic assessment of the performance of the chosen mapping. This corrected measure of the total execution time is the time that appears in the results. The reevaluation of the chosen mapping is done only after the mapping algorithm has terminated and only for the sake of comparison.

- **YBN Modified** The mapping algorithm described above, **YBN** is modified to check for potential channel contention. This is used to compare our heuristic for cluster assignment with the **YBN** heuristic.

- **Random** A random mapping is generated. The execution time of this mapping is determined through simulating its execution.

The figures and tables present the average of 100 randomly generated task graphs. The results of each simulation are normalized relative to the execution time of that task graph on a completely connected architecture. Hence, the value of 100 represents the execution time on a completely connected architecture.

Tables 3 – 5 and Figures 31 – 32 clearly show that a better mapping is chosen when channel contention is considered. Table 3 shows how the relative performance increases with the number of processors if the number of tasks per processor is fixed. With fifty tasks per processor, the relative improvement of our heuristic increases from

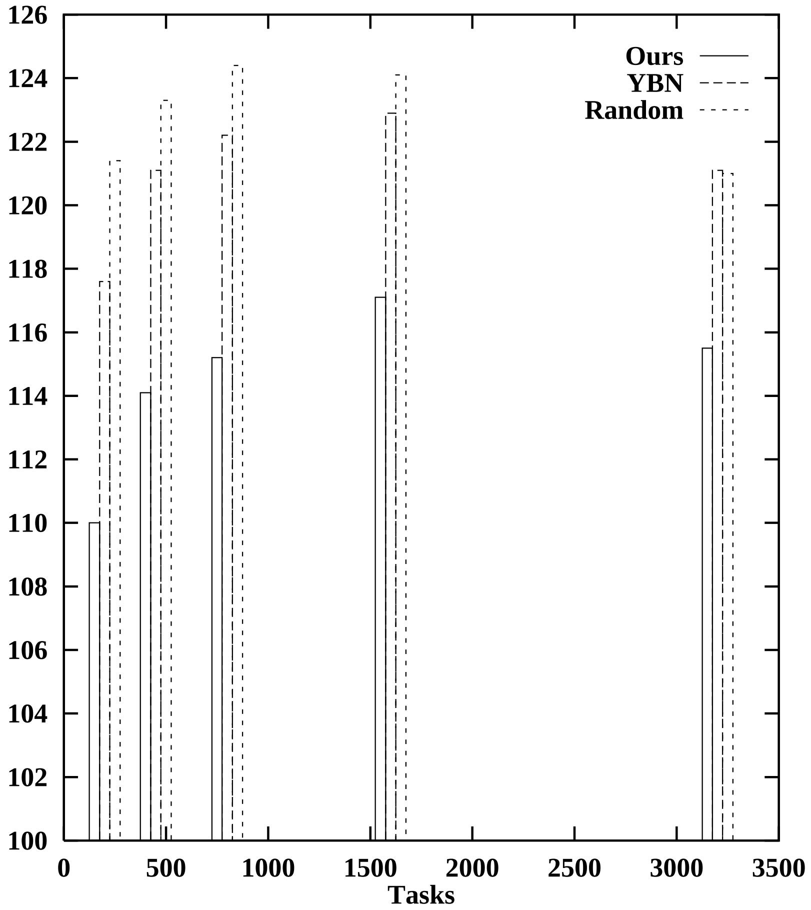


Figure 31: Relative Execution Times for 16 Processors

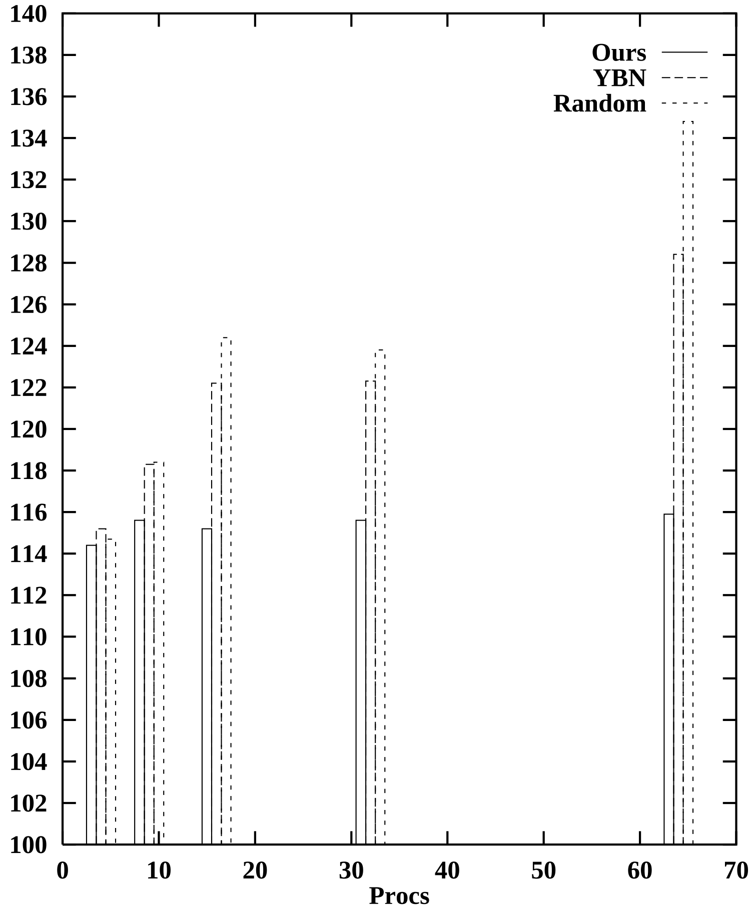


Figure 32: Relative Execution Times for 800 Tasks

Table 3: Relative Execution Times

Processor & Problem Size	Ours No B/W	Ours	YBN	YBN Modified	Random
p = 4, V = 200	113.3	112.1	113.0	113.4	113.2
p = 8, V = 200	116.4	112.3	117.4	117.4	117.5
p = 8, V = 400	117.5	113.8	118.1	117.8	117.6
p = 16, V = 400	120.8	114.1	121.1	118.1	123.3
p = 16, V = 800	121.3	115.2	122.2	118.8	124.4
p = 32, V = 800	122.3	115.6	122.3	120.0	123.8
p = 32, V = 1600	129.1	121.5	131.3	123.0	134.5
p = 64, V = 1600	134.4	123.5	138.5	126.3	142.6
p = 64, V = 3200	143.4	131.6	145.8	133.9	150.1

Table 4: Relative Execution Times Varying the Number of Tasks

Processor & Problem Size	Ours No B/W	Ours	YBN	YBN Modified	Random
p = 16, V = 200	116.1	110.0	117.6	112.3	121.4
p = 16, V = 400	120.8	114.1	121.1	118.1	123.3
p = 16, V = 800	121.3	115.2	122.2	118.8	124.4
p = 16, V = 1600	122.2	117.1	122.9	121.7	124.1
p = 16, V = 3200	119.6	115.5	121.1	119.7	121.0

about 1% to more than 10%. With twenty-five tasks per processor, the improvement ranges from nearly 5% to more than 12%. As shown in Table 4 and Figure 31, the relative performance improvement is almost constant at about 5-7% when the number of processors is fixed and the number of tasks is increased. Although the relative performance improvement is nearly constant, the absolute performance improvement increases, because the actual execution time increases. Table 5 and Figure 32 show

Table 5: Relative Execution Times Varying the Number of Processors

Processor & Problem Size	Ours No B/W	Ours	YBN	YBN Modified	Random
p = 4, V = 800	115.3	114.4	115.2	115.1	114.7
p = 8, V = 800	118.0	115.6	118.3	118.4	118.4
p = 16, V = 800	121.3	115.2	122.2	118.8	124.4
p = 32, V = 800	122.3	115.6	122.3	120.0	123.8
p = 64, V = 800	125.0	115.9	128.4	118.8	134.8

how the performance difference *increases* with the number of processors from about 1% with four processors to more than 10% with sixty-four processors.

Ignoring the delay caused by channel contention results in a less efficient mapping. The difference between the heuristic proposed in [102] and the modified version that considers contention is smaller than the difference between our heuristic and YBN, because the critical clusters cannot be reassigned. The difference is always less than 9%. Frequently, especially for multiprocessors with only a few processors, all the clusters are critical. For those cases, the algorithm in [102] tries only one mapping.

The reassignment of the critical clusters provides the mapping algorithm with greater flexibility in creating mappings. Since the number of possible mappings is greater, the likelihood of finding a better mapping increases. This is clearly seen by comparing our algorithm with the modified version of the algorithm in [102]. Although the improvement is slight, always less than 4%, our heuristic is consistently better than YBN even when YBN is modified to consider contention. Our mapping heuristic is not always better than YBN when our heuristic is modified to ignore contention, because the best mapping is being chosen using inaccurate assumptions.

Except for the smallest task graphs on the smallest hypercubes, the random mapping algorithm has the worst performance. This is not surprising, since the probability that the random mapping generates an optimal mapping decreases with more processors. The variation between the results in this paper is much less than the variation shown in [102]. The reason for this relatively limited variation is that packet switching is used. Message switching is used in [102]. Message switching is more sensitive than packet switching to the distance between processors. Since a random mapping is more likely to have critical edges that are not transmitted between neighboring processors, the communication overhead for the random mapping is significantly higher with message switching than with packet switching.

Although our heuristic generally finds a better mapping than the heuristic in [102], there are infrequent cases when our algorithm does not find a better mapping. We have found two reasons for this. First, both heuristics attempt to reduce the communication overhead of the critical edges. Because different heuristics are used, it is possible that the heuristic in [102] inadvertently chooses a mapping that results in less channel contention. Second, a trade-off exists between mapping critical clusters close together to decrease the transmission time of the critical edges and mapping the critical clusters further apart to allow a larger number of channel-disjoint paths between the source and destination processors. When two clusters are on neighboring processors, there is only one shortest path between the two clusters. If more than one message is being transmitted between these two clusters at the same time, one of the messages must be delayed. In this case, it would be better to place the clusters further apart! Since the critical edges tend to be closer with our heuristic, it suffers more from this effect. Note that in both cases, the heuristic proposed in [102] is better due to chance, not some specific attribute of their heuristic.

Other than the random mapping, the greatest difference is consistently between our algorithm and the algorithm proposed in [102]. This provides additional support for our premises that an accurate model of communication and the continual refinement of the set of critical edges produce a more efficient mapping.

5.4 Summary

In this chapter, we have presented a heuristic for the compile-time mapping of clustered task graphs. This heuristic focuses on reducing channel contention. The simulation results are presented for mapping random DAGs on a packet-switched hypercube which uses fully adaptive routing. In Chapter VI, we compare these two heuristics for DAGs derived from the Cholesky factorization of sparse matrices. In addition, we apply the heuristics to wormhole-routed meshes and hypercubes. Thus, we can use the routing algorithms proposed in Chapter III and Chapter IV. The simulations performed in Chapter VI combine our results on adaptive routing and mapping into a unified result.

CHAPTER VI

Integrating Mapping and Routing

In this chapter we present a simulation study of a comprehensive approach to reducing contention in distributed-memory multiprocessors. In the dissertation, our primary focus has been adaptive routing. In Chapter III we presented an optimal fully adaptive routing algorithm for meshes and in Chapter IV we presented a fully adaptive hypercube routing algorithm that is substantially more adaptive than any previously known. We have also addressed the problem of channel contention with the mapping heuristic proposed in Chapter V. Since adaptive routing and mapping are complementary approaches to reducing contention, a combination of both techniques should lead to a better result than using only one of the techniques. The mapping heuristic should generate different mappings for each routing algorithm, because the channel contention is affected by the routing algorithm. In addition, the mapping heuristic should produce better mappings than a heuristic that ignores channel contention. Furthermore, adaptive routing should lead to better performance, although the mapping heuristic may reduce the inherent advantage of adaptive routing for reducing contention – since the mapping heuristic selects the best mapping for each routing algorithm individually, the contention should be reduced for all the routing algorithms, including nonadaptive routing.

In Section 6.1,the DAGs used in the simulation study are described and the clustering method is explained in Section 6.2. The simulation methodology is then described in Section 6.3. Simulation results for hypercubes and 2D meshes are presented in Section 6.4. The simulation results are discussed as they are presented. Section 6.5 has some concluding remarks about the simulation results.

6.1 DAG Characteristics

Rather than simulating random graphs, we use DAGs derived from the Cholesky factorization of three different sparse matrices arising in real applications. To provide differing ratios of computation to communication, CPU clock cycles of 20ns and 5ns are used. DAGs 1, 3, and 5 have a clock cycle of 20ns, and DAGs 2, 4, and 6 are DAGs 1, 3, and 5, respectively, with a 5ns clock cycle. The computation and communication characteristics of each DAG are summarized in Table 6. The task weight in number of instructions and the message length in bytes are shown in Table 6, also shown are the number of tasks and messages in the DAG. The message length does not include the header, which is prepended during simulation. The task weights and message lengths have been proportionally reduced from those in the original DAG, thereby reducing the simulation time while maintaining the original DAG structure. Since the task weights and message lengths have been proportionally reduced, the message startup times must also be reduced. For this reason, the startup time of a message is ten CPU clock cycles per message. If a task sends more than one message, these messages are injected at intervals of ten clock cycles to simulate the startup time of each message. The communication characteristics change when the number of processors changes, because the cluster merging reduces the number of inter-processor messages and more merging is done when there are fewer processors. The number and length of the inter-processor messages are shown in Table 7.

Table 6: DAG Computation and Communication Characteristics

DAGs	# of Tasks	Task Weight Min/Avg/Max	# of Msgs	Msg Length Min/Avg/Max
1 & 2	469	9/13.0/21	564	14/19.1/26
3 & 4	659	9/22.2/45	915	14/29.4/50
5 & 6	3839	9/14.4/33	4878	14/21.6/38

6.2 Generating and Processing Clusters

Our mapping heuristic assumes that the input DAG is a clustered task graph with one cluster per processor. We have used the Dominant Sequence Clustering (DSC) heuristic proposed by Yang and Gerasoulis [105] to cluster the task graphs and the cluster merging algorithm used in the Pyrros [104] system to produce the correct number of clusters. This cluster merging algorithm was proposed by George, Heath, and Liu [35] and merges clusters based on their computation characteristics, so the communication structure of the clustered DAG is ignored during the merging step.

Table 7: Inter-Processor Communication Characteristics

Input		Injected Msgs	
DAG #	# of Procs	# of Msgs	Msg Length Min/Avg/Max
1	16	399	14/18.6/26
1	64	429	14/18.9/26
2	16	441	14/18.9/26
2	64	468	14/19.0/26
3	16	730	14/29.2/50
3	64	795	14/28.8/50
4	16	766	14/29.2/50
4	64	811	14/29.0/50
5	16	3492	14/21.0/38
5	64	3783	14/21.2/38
6	16	3742	14/21.3/38
6	64	3904	14/21.3/38

Once the clustering and merging steps have been performed, the number of clusters matches the number of processors. Unlike the assumption made in Chapter V, the tasks in a cluster do not have a total ordering. Hence, 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 injected into the network in increasing order of the latest starting time of the destination tasks.

6.3 Simulation Methodology

The mapping heuristic simulates the performance of a mapping to accurately compute the execution time of the mapping and to determine the channel contention.

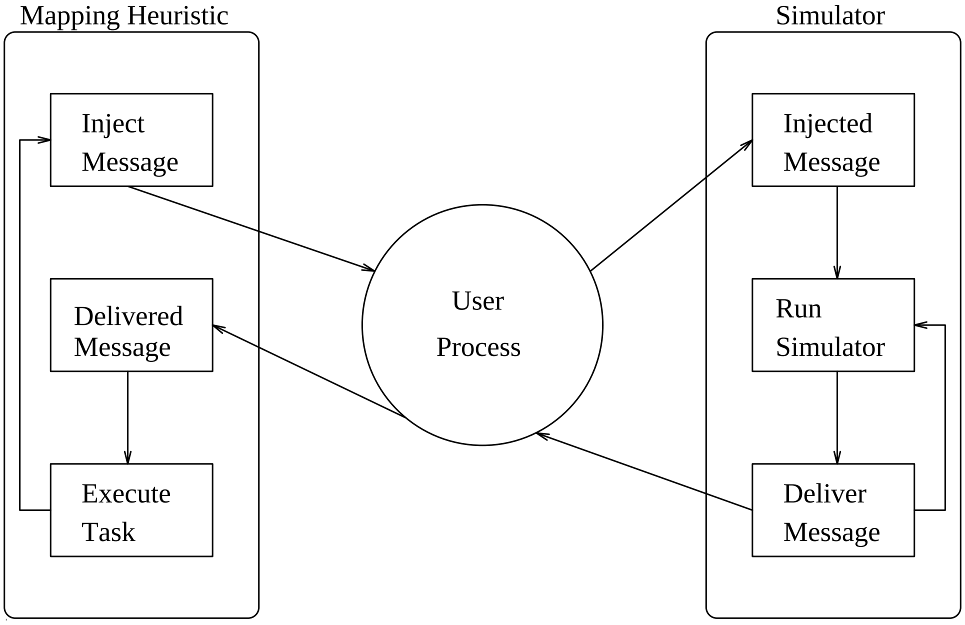


Figure 33: Interaction between the Mapping Heuristic and the Simulator

The simple simulator used in Chapter V has fairly limited capabilities. Rather than extending this simulator, we used a simulator [47] developed using the CSIM [87] simulation package. The modular structure of the simulator allows the specification of different multiplexing schemes, routing algorithms, network topologies, and message traffic patterns. All the experiments in this chapter use demand multiplexing. Both hypercube and 2D meshes are used, with a variety of routing algorithms.

The simulator allows different message traffic patterns to be specified by modifying a single process, called the *user process*. The simulation is event-driven, with each message corresponding to a unique event. User processes have been written to support uniform traffic, hot spot traffic, transpose traffic, etc. For these simulations, however, the user process must support message traffic that is generated from the DAG that is being scheduled with the mapping heuristic. This requires interaction between the mapping heuristic and the simulator as depicted in Figure 33, which shows the high-level operation of the user process for these simulations. In essence, the user process functions as the network interface depicted in Figure 1(b) on page 4. When a task completes execution, the messages sent by this task are injected into the network. The mapping process injects messages by writing these messages into shared memory. 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 mapping process. The mapping process uses the delivery of a message to initiate another task (if a task is ready to run) and then injects additional messages into the network.

Initially, the clusters are assigned to the processors using the identity mapping. The interaction between the simulator and the mapping heuristic allows the contention experienced by messages to be relayed to the mapping heuristic. The mapping is then modified to reduce the contention. Since different routing algorithms experience different channel contention patterns, the mapping is adjusted to match the particular characteristics of each routing algorithm.

The mapping heuristic and the simulator are run as separate Unix processes. In order to support the interaction between these two processes, the mapping heuristic was modified to use the shared memory facilities of Unix. In addition, a new user process was written that would read injected messages from shared memory and write delivered messages into shared memory. Two producer-consumer queues are defined in the shared memory block, one for injecting messages into the network and one for delivering messages to the processor.

As previously mentioned, adaptive routers are more complex than nonadaptive routers [8]. The simulator achieves greater accuracy by allowing the network cycle time to be specified by the user. The clock cycle times for the topologies and routing algorithms we have simulated are described in Table 8. The clock cycle times are derived from the cost model used by Patel and Jayasimha [80], which is based on Chien's model [8]. The calculation of network cycle time based on router complexity permits a more balanced comparison of the performance of different routing algorithms. The routing algorithms are implemented with two virtual channels per physical channel. Both opt-y and mad-y require only a single channel in the X dimension and two virtual channels in the Y dimension. The router cycle time for 2D mesh with this set of virtual channels is 5.70ns. For XY routing, we assume different crossbars for the X and Y dimensions, so the router design has been optimized for XY routing. All other routing algorithms use a single crossbar.

Table 8: Router Cycle Times (in ns)

Routing	Hypercubes		Meshes	
Algorithms	4D	6D	4×4	8×8
Opt-y	NA	NA	5.92	5.92
Mad-y	NA	NA	5.92	5.92
XY	NA	NA	5.41	5.41
Enhanced	5.92	6.24	NA	NA
Duato	5.92	6.24	NA	NA
e-cube	5.92	6.24	NA	NA

6.4 Simulation Results

The simulations in this chapter are for wormhole-routed multiprocessors. Results are presented for both meshes and hypercubes. The fully adaptive routing algorithms we have proposed are compared with the previous best known fully adaptive routing algorithms for these topologies. Nonadaptive routing is also used for comparison.

The execution time of each DAG (in nanoseconds) is presented. Except for Enhanced Fully Adaptive and Duato's algorithm, the routing algorithms provide more than one waiting channel for a blocked message. When a message is blocked, the simulator randomly selects a waiting channel from the set of waiting channels (otherwise, the simulations are completely deterministic). This leads to small variations in the execution times reported by the mapping heuristic – hence, the reported execution time is the average of three simulations.

6.4.1 Hypercube Simulations

The hypercube simulation results compare our Enhanced Fully Adaptive routing algorithm with Duato's fully adaptive routing algorithm and the nonadaptive *e*-cube routing algorithm. The simulations were run for 4D and 6D hypercubes and the results presented are the average of three runs. Since a single crossbar design for a router is assumed for all three routing algorithms, the router cycle times are the same for both nonadaptive routing and both adaptive routing algorithms.

The results in Tables 9 and 10 show that the smallest four DAGs do not scale well from 16 to 64 processors. One reason is that contention increases with the size of

Table 9: 4D Hypercube Results (in ns)

	Our Mapping Heuristic			Yang, Bic, and Nicolau		
DAG	Enhanced	Duato	*e*-cube	Enhanced	Duato	*e*-cube
1	15683	15628	16325	15776	15776	16820
2	38033	38033	38222	39430	39430	39700
3	20663	20948	21748	22247	21383	23092
4	55912	55912	55746	57931	57931	58119
5	49270	49430	51689	51613	50197	56783
6	142705	144407	141875	146595	146595	146793

Table 10: 6D Hypercube Results (in ns)

	Our Mapping Heuristic			Yang, Bic, and Nicolau		
DAG	Enhanced	Duato	*e*-cube	Enhanced	Duato	*e*-cube
1	15999	15958	16275	16145	16145	16613
2	37119	37119	37049	37630	37630	37580
3	19010	18767	20101	20023	20267	20947
4	48449	48449	48547	49286	49286	49683
5	34103	34646	36827	35847	35754	38039
6	81678	81934	83735	87758	87758	85931

the interconnection network. Furthermore, a few processors have only a single task assigned to them and the overhead of transmitting messages to and from this processor are greater than the reduction in computation achieved by using this processor. DAGs 5 and 6 are more scalable, since there are more tasks to distribute among the processors.

Our mapping heuristic always performs better than the YBN heuristic proposed in [102]. This supports our claim that ignoring channel contention leads to inferior mappings, especially with wormhole routing, where latency is almost independent of distance.

As expected, the adaptive routing algorithms consistently perform better than the nonadaptive routing algorithm. Nonadaptive routing is better than adaptive routing only for the YBN mappings of DAG 2 and DAG 6 on the 6D hypercube, and for DAGs 4 and 6 with our mapping heuristic. Although our mapping heuristic generates a better mapping than the YBN heuristic, our mapping heuristic does not necessarily produce a mapping of equal quality for each routing algorithm. Different mappings are generated for nonadaptive and adaptive routing, so, on occasion, the performance of nonadaptive routing may be better. For the same reason, with our mapping heuristic, Duato's routing algorithm performs better than Enhanced for DAG 1 on the 4D hypercube and DAGs 1 and 3 on the 6D hypercube. Duato's routing algorithm also performs better with the YBN heuristic for DAGs 3 and 5 on the 4D hypercube and for DAG 5 on the 6D hypercube. Otherwise, the performance of the Enhanced Fully Adaptive routing algorithm is either better or the same. The average distance on an nD hypercube is only about $n/2$. The mapping heuristic should further reduce the average distance of a message. Each DAG has relatively few messages that are critical edges. As long as these edges are delivered at the same time and the other edges are not delayed too much, the performance of the two fully adaptive routing algorithms will be the same. Furthermore, an increase in the message latency for a few critical edges can result in a substantial increase in the execution time with a given routing algorithm, even if the average message latency is lower than the other routing algorithms. In order to examine this hypothesis, we have measured the average message latency for some of the DAGs. The average message latency is lowest with the Enhanced Fully Adaptive routing algorithm, even when the performance is better with other routing algorithms.

Furthermore, the nonadaptive routing algorithm has one advantage over the adaptive routing algorithms; it has two waiting channels for a message, while the adaptive routing algorithms have a single waiting channel. When the network becomes congested, the nonadaptive routing algorithm can distribute blocked messages over two channels and the average latency is thereby decreased. On the other hand, the adaptive routing algorithms queue up all the blocked messages on a single waiting channel. This problem is somewhat lessened because not all messages need to route in the same dimensions. Even so, the average queue length with adaptive routing should be about double the average queue length with nonadaptive routing. The distribution of the waiting channels and the low diameter of the 4D and 6D hypercubes explain why the performance difference between nonadaptive and adaptive routing is usually modest.

6.4.2 2D Mesh Simulations

Table 11 presents the simulation results for a 4×4 mesh with two virtual channels per physical channel. The corresponding results for an 8×8 mesh are shown in Table 12. Since opt-y and mad-y require only one virtual channel in the X dimension, simulation results are presented both when the routing algorithms have two virtual channels per physical channel and when only one channel is used in the X dimension. We first describe the simulation results with two virtual channels in the X dimension.

Table 11: 4×4 Mesh Results (in ns)

	Our Mapping Heuristic			Yang, Bic, and Nicolau		
DAG	Opt-y	Mad-y	XY	Opt-y	Mad-y	XY
1	16466	16525	15732	17275	17162	16441
2	38443	38789	38003	39212	38877	38647
3	23733	24112	23654	27021	27633	26149
4	56482	56016	55538	57846	57168	56705
5	71061	71099	63026	71812	76813	64660
6	143511	144088	143308	147977	146879	146579

Table 12: 8×8 Mesh Results (in ns)

	Our Mapping Heuristic			Yang, Bic, and Nicolau		
DAG	Opt-y	Mad-y	XY	Opt-y	Mad-y	XY
1	16198	16404	15449	17159	17267	16457
2	37236	37193	36596	38034	38034	37355
3	21035	20946	20334	22198	22339	22086
4	48883	48866	48247	50090	49818	49108
5	48662	49894	44824	58385	58846	49708
6	85416	86651	84070	89549	92201	86854

As with the hypercubes, our mapping heuristic always produces a better mapping than the YBN mapping heuristic. Our mapping heuristic is better for all three routing algorithms on all six DAGs. Our mapping heuristic is able to generate more efficient mappings because the heuristic measures channel contention and adjusts the mapping to reduce contention. For DAG 5 on an 8×8 mesh, the improvement is over 20%, which is somewhat surprising since the communication is relatively sparse. As with the hypercubes, the four smallest DAGs have limited scalability, while DAGs 5 and 6 are more scalable.

Our mapping heuristic consistently chooses different mappings for each of the routing algorithms. Thus, we are able to generate mappings that are specific to a routing algorithm and address the contention characteristics of that routing algorithm. A comparison of the routing algorithms for the mapping generated using the YBN mapping heuristic shows that the performance difference between the nonadaptive routing algorithm and the adaptive routing algorithms is usually less with the YBN heuristic, because the same mapping is being used for all three routing algorithms. Because of the lower clock cycle time, nonadaptive routing is best when the routing algorithms use the same mapping. This is because their mapping heuristic ignores contention, and hence, does not choose the best mapping based on the routing algorithm. The additional message latency that the critical edges experience is not reduced for subsequent mappings.

XY consistently performs better than both adaptive routing algorithms. The primary cause of this phenomenon is the increased network cycle time for adaptive routing. We show that the increase in cycle time accounts for most of the difference in performance. Tables 13 and 14 show the execution times, ignoring the router complexity, and consequently, using the same cycle time for all the routing algorithms. In fact, the adaptive routing algorithms now often perform better than nonadaptive routing. When the nonadaptive routing is better, the difference is much less than the difference shown in Tables 11 and 12.

Table 13: 4×4 Mesh with Identical Cycle Times (in ns)

	Our Mapping Heuristic			Yang, Bic, and Nicolau		
DAG	Opt-y	Mad-y	XY	Opt-y	Mad-y	XY
1	15673	15440	15870	16281	16019	16400
2	38175	37713	37889	38442	38914	38672
3	22437	22667	23632	24135	24676	25968
4	55748	55341	55658	57115	56731	56808
5	65078	66509	63724	69429	69705	67498
6	141967	138640	140636	146840	147148	145193

Table 14: 8×8 Mesh with Identical Cycle Times (in ns)

	Our Mapping Heuristic			Yang, Bic, and Nicolau		
DAG	Opt-y	Mad-y	XY	Opt-y	Mad-y	XY
1	15333	15417	15516	16334	16087	16444
2	36588	36375	36462	37240	37247	37340
3	19978	19674	20007	20905	21055	22504
4	48267	47884	48158	49193	49262	49066
5	47018	46403	44659	54861	54524	52984
6	83793	83887	84314	87154	88888	87018

The comparison of the adaptive routing algorithms with the original cycle times shows that opt-y performs better than mad-y for about two-thirds of the DAGs. In the cases where mad-y performs better, the difference is usually negligible and never more than 2%. On the other hand, opt-y is as much as 7% better than mad-y. When identical cycle times are used, there is less difference between opt-y and mad-y, because the faster network cycle time results in less contention.

Besides the faster network cycle time, there are two other reasons why nonadaptive routing can perform better than adaptive routing. First, our mapping heuristic maps clusters with critical edges as close together as possible. This helps reduce contention, but also limits the usefulness of adaptive routing, since there are fewer routes for avoiding contention. This is particularly problematic when there are multiple messages between the same two processors. (An example is the first situation resulting in channel contention as shown in Figure 28 on page 152.)

Second, the total execution time depends on the message latency of the relatively few messages that are critical edges. For a given mapping, even if the adaptive routing algorithms achieve a lower average message latency, the latency of the critical edges could be lower with nonadaptive routing. Since the mapping heuristics do not explicitly address this issue, nonadaptive routing can perform better by chance.

Since mad-y and opt-y both require only one virtual channel in the X dimension, we also simulated meshes with this configuration. The results are presented in Tables 15 and 16. As discussed when presenting Table 8, the network cycle times for this configuration are lower for the adaptive routing algorithms but unchanged for XY. Thus, the execution times for adaptive routing are closer to the execution time for nonadaptive routing. Otherwise, the results are comparable to the results shown in Tables 11 and 12. The performance of opt-y and mad-y are comparable in most cases and which is better depends on the DAG and the number of processors. When the routing algorithms are used on the mapping generated by the YBN heuristic, mad-y is slightly better than opt-y in some cases. Invariably, when the performance difference is more than 1%, opt-y is the better of the two.

Table 15: 4×4 Mesh Results with 1 VC in X (in ns)

	Our Mapping Heuristic			Yang, Bic, and Nicolau		
DAG	Opt-y	Mad-y	XY	Opt-y	Mad-y	XY
1	15421	15277	15018	16121	16230	15542
2	37986	37973	37824	38923	38871	38699
3	24492	24470	24467	27956	27529	26618
4	54952	55886	55018	57018	57006	56169
5	78893	78690	68913	81516	82715	75519
6	140943	141078	141745	146884	146520	145877

Table 16: 8×8 Mesh Results with 1 VC in X (in ns)

	Our Mapping Heuristic			Yang, Bic, and Nicolau		
DAG	Opt-y	Mad-y	XY	Opt-y	Mad-y	XY
1	15036	14997	14598	16423	16308	16027
2	36798	36828	36512	37616	37570	37213
3	20106	20232	20322	22058	22071	25832
4	48289	48170	47927	49241	49856	49292
5	54629	55644	47668	65989	65913	57222
6	86788	87333	85383	94349	99702	88171

6.5 Simulation Summary

We have presented simulation results of a comprehensive approach to the problem of contention. The simulation results show that our mapping heuristic is always better than the YBN heuristic. Because contention is the major cause of increased message latency with wormhole routing, ignoring contention leads to inferior mappings. We have also shown that, with a uniform network cycle time, adaptive routing is generally better than nonadaptive routing. For meshes, we have tended to favor nonadaptive routing by using optimized router designs for XY routing and unoptimized router designs for the adaptive routing algorithms. Efficient designs for the adaptive routers will decrease the difference in router cycle times and should lead to better performance for adaptive routing.

The interaction between adaptive routing and mapping can affect the performance of the routing algorithms. Mapping critical clusters onto adjacent processors reduces the advantages of providing multiple paths between the source and destination. Thus, another possibility for improving the relative performance of adaptive routing is the design of more intelligent mapping heuristics.

The lack of consistent results among the routing algorithms suggests that our mapping heuristic does not adequately address the problem of contention for wormhole-routed multiprocessors. In other words, by indirectly reducing contention by decreasing distance, the mapping heuristic may not reduce contention in ways that maximize the performance of the routing algorithms. Hence, even if the average message latency is good, the routing algorithm can perform poorly if the mapping increases the message latency of a few critical edges.

We have explored the interaction between mapping and adaptive routing with only a small set of DAGs. To do a more comprehensive comparison, we need to consider (a) other types of DAGs and (b) a mix of jobs concurrently sharing the multiprocessor. For the latter situation, a lower average latency should have a more significant effect on execution time than the effect shown in these simulations. In addition, one consequence of achieving deadlock freedom with routing restrictions is that adaptive routing introduces imbalance in the network by favoring certain message routes. The mapping heuristic may realize better performance by exploiting this imbalance. These issues are discussed further in Chapter VII.

CHAPTER VII

Conclusions

This dissertation studies the problem of contention in distributed-memory multiprocessors. Since channel contention is a major component of increased message latency in current multiprocessors, the performance of parallel processors can be improved by reducing contention. Our results make contributions to two different techniques for reducing contention: an architectural approach – adaptive routing, and a software mechanism – mapping. The primary emphasis in this dissertation has been adaptive routing. We have posed and answered several fundamental questions related to the optimality and deadlock freedom of adaptive routing algorithms. The proposed architectural and software solutions have also been integrated via simulation.

In this chapter, our contributions are summarized and possible extensions to our work are described. We also discuss the future applicability of these results, given the rapid changes in VLSI technology, which in turn affect the design of parallel computer architectures.

7.1 Research Contributions

Chapters III, IV, and V present our main contributions, with a fourth contribution being the integrated approach presented in Chapter VI. Each contribution is described separately.

7.1.1 An Optimal Mesh Routing Algorithm

An optimal fully adaptive wormhole routing algorithm for n-dimensional meshes has been presented in Chapter III. This result makes theoretical contributions by proving a tight lower bound on the number of virtual channels required for fully adaptive routing and a tight lower bound on the number of restrictions required on the use of this set of virtual channels. In addition, we have proved that any other routing algorithm that is optimal in both ways *must* be a symmetric case of the routing algorithm we have proposed.

Besides the theoretical contributions, the routing algorithm has several important practical advantages. The routing algorithm is more adaptive, and requires fewer virtual channels, than any other fully adaptive mesh routing algorithm that has been proposed. If extra virtual channels are available, our optimal routing algorithm can be easily extended to utilize these additional virtual channels with the minimum number of restrictions.

7.1.2 A Universal Proof Technique

Chapter IV presents the first generally applicable necessary and sufficient condition for proving deadlock freedom. One feature of our methodology is the introduction of two concepts: the *channel waiting graph* and *False Resource Cycles*. These two concepts support a framework for generating simple proofs of deadlock freedom. Our methodology is quite developed, with a formal procedure for distinguishing between False Resource Cycles and True Cycles, and an algorithm for reducing the *channel waiting graph* to an appropriate CWG'.

By characterizing the problem of deadlock freedom in terms of channel waiting instead of channel usage, the proofs of deadlock freedom become natural and straightforward. For nonadaptive routing, the channel waiting graph and the channel dependency graph are essentially equivalent. For adaptive routing, however, the use of the channel waiting graph avoids the difficulties of extending the channel dependency graph to adaptive routing algorithms, where the choice of output channels introduces additional possibilities for avoiding deadlock. For example, Duato's methodology requires the identification of a suitable routing subfunction and the consideration of four different kinds of channel dependencies. This illustrates the disadvantage of extending the channel dependency graph to adaptive routing algorithms.

The idea of waiting channels is applicable not only to wormhole routing, but also to store-and-forward routing and virtual cut-through. In fact, any switching technique that relies on routing restrictions to prevent deadlock can use our methodology. Thus, our proof technique is universal. Other than using different terminology, such as buffer instead of channel, the proofs of the necessary and sufficient condition are unchanged. The switching technique affects the structure of the channel waiting graph, but the underlying idea of waiting dependencies is independent of the switching technique.

For example, with store-and-forward routing, a packet can occupy only one adjacent buffer while waiting for an output buffer. Our definition of waiting channel (waiting buffer) recognizes this fact, by defining waiting buffers based on the relationship between occupying a buffer and waiting for a buffer.

A procedure is provided for recognizing False Resource Cycles and for reducing the CWG to CWG'. In practice, however, neither of these procedures is likely to be required and a simple inspection of the routing algorithm and the topology should be sufficient.

Our methodology applies to a larger class of routing algorithms than previous necessary and sufficient conditions and this translates into a more applicable result. For example, the proof of theorem 4.6 would not be possible with Duato's methodology, because the Enhanced Fully Adaptive routing algorithm is incoherent.

We have used our necessary and sufficient condition to propose a fully adaptive hypercube routing algorithm for wormhole routing, the Enhanced Fully Adaptive routing algorithm, which is substantially more adaptive than previous routing algorithms. We have shown that relaxing the restrictions on the routing algorithm results in deadlock. Furthermore, we have shown that the minimum number of restrictions is imposed. These proofs were greatly facilitated by the necessary and sufficient condition.

7.1.3 A Contention-Sensitive Mapping Heuristic

In Chapter V, we have presented a mapping heuristic for clustered task graphs. The mapping heuristic considers channel contention when mapping the task graph and adjusts the mapping to reduce channel contention. Ignoring contention can result in a significant performance penalty by selecting inferior mappings. Early mapping heuristics that focused on reducing distance rather than contention are unlikely to perform significantly better than random mappings.

We have used our heuristic with packet switching. The effect of contention on packet-switched networks is somewhat surprising, given the impact of distance on message latency. The impact of contention should be even greater for wormhole routing and virtual cut-through. The widespread use of wormhole routing and virtual cut-through in current multiprocessors suggests that reducing channel contention should be the focus of mapping heuristics used on current commercial parallel processors.

7.1.4 A Comprehensive Approach

The architectural mechanism, adaptive routing and the software mechanism, mapping, that we have used to address contention in distributed-memory multiprocessors can be used in conjunction. We have used a simulator developed with CSIM to model the interconnection network and routing algorithm, and have used Unix shared memory constructs to allow our mapping heuristic to interact with the simulator. The mapping heuristic generates the message traffic for the simulator and the simulator uses various routing algorithms to transmit messages. This environment has permitted us to provide a more realistic model of a multiprocessor architecture and has allowed the use of message traffic characteristics from real applications.

This comprehensive approach has given us several insights into the performance of different routing algorithms and mapping heuristics when scheduling task graphs from real applications. Although a more detailed study, consisting of many task graphs, would be valuable, some trends are clear. First, our mapping heuristic is consistently better than the mapping heuristic proposed by Yang, Bic, and Nicolau, because channel contention is considered. Second, our fully adaptive routing algorithms perform comparable to, and often better than, previous fully adaptive routing algorithms, however, both routing algorithms suffer from the limitations of the mapping heuristics. The total execution time depends on the latency of a small proportion of the messages. Even if the *average* message latency is better with adaptive routing, a few messages may have higher latency with adaptive routing than nonadaptive routing. If the critical messages experience additional latency, the execution time of the DAG is effected. Mapping heuristics that better address the issue of contention are required to exploit the advantages of adaptive routing.

7.2 Future Work

General results have been presented in this dissertation and these results can be naturally extended in many different ways. In this section, we describe some of the more immediate possibilities for future work.

7.2.1 Routing Algorithms for Other Topologies

In this dissertation, routing algorithms for meshes and hypercubes have been proposed. The Cray T3D has generated renewed interest in routing algorithms for the torus. In addition, routing algorithms for de Bruijn networks and octagonal meshes, for example, are of theoretical interest and perhaps may have practical application in future multiprocessor architectures.

Our methodology based on the necessary and sufficient condition can be applied to any topology. Since we have been able to generate simple straightforward proofs of deadlock freedom for the routing algorithms we have proposed, the application of the technique to these, and other, topologies should also be relatively simple. This is an interesting opportunity for extending the work in this dissertation.

7.2.2 Proving Optimality for Other Topologies

An optimal fully adaptive routing algorithm is an open problem for *every* topology except n-dimensional meshes. For hypercubes, we have partially addressed this question. The only remaining requirement for proving that the Enhanced Fully Adaptive routing algorithm is optimal is to prove that the restrictions cannot be further relaxed. Potentially, this requires an evaluation of a broad class of routing algorithms. The availability of the necessary and sufficient condition should simplify this evaluation, because deadlock could be proved for alternative routing algorithms without constructing specific examples of deadlock configurations. Of course, the Enhanced Fully Adaptive routing algorithm may not be optimal.

The necessary and sufficient condition provides an important tool for addressing the issue of optimal routing algorithms. Using the ideas presented in Chapter III and Chapter IV, optimal proofs for other topologies could be derived. In other words, intelligent arguments for restricting the set of routing algorithms to consider seems to be a reasonable first step. The necessary and sufficient condition can then be used to rule out possibilities on a case-by-case basis until optimality is shown. It is not obvious how to restrict the set of possibilities to a reasonable number for an arbitrary topology. This is another opportunity for extending the dissertation research.

7.2.3 Proving Optimality for Other Switching Techniques

The necessary and sufficient condition we have proposed also applies to virtual cut-through and store-and-forward routing. A natural extension of this research is the design of optimal routing algorithms for these switching techniques.

7.2.4 Identifying False Resource Cycles

The method we have proposed for identifying False Resource Cycles is a complete solution only for routing algorithms that are suffix-closed. For routing algorithms that are not suffix-closed, it may be possible to generate cycles where the shared channel is not part of the cycle. This is an interesting open problem with two different possibilities. One resolution would be to describe an algorithm that determines when simultaneous use of the channel is required for a deadlock configuration. The second resolution would be to show that any cycle where the shared channel is outside the cycle becomes a True Cycle with sufficiently small messages. If the latter resolution is proved, then only channels within the cycle must be considered and our approach becomes a complete solution even for routing algorithms that are not suffix-closed. This question may be primarily of theoretical interest, because other than simple examples proposed specifically to demonstrate this possibility, we are unaware of any routing algorithms that are not suffix-closed.

7.2.5 Fault-Tolerant Routing

Our necessary and sufficient condition can also be used to propose fault-tolerant routing algorithms. The partially adaptive mesh routing algorithm proposed in Chapter IV uses our methodology to achieve more nonminimal (as well as minimal) adaptiveness. Supporting more nonminimal routing improves the fault-tolerance by providing more nonminimal paths around faulty nodes and channels. Routing algorithms that combine a high degree of adaptiveness, fault tolerance, and simple router designs are needed to support massively parallel processing. The area of fault-tolerant routing presents an additional opportunity for building on this work.

7.2.6 Better Mapping Heuristics

Existing mapping heuristics, even those that focus on reducing channel contention, typically proceed by trying to move communicating tasks to neighboring processors. The underlying idea is that sending messages over a shorter distance decreases the chance of experiencing contention. This is often true, however, there are cases where longer paths result in less contention. An example is mapping tasks to the processors on the edge of a mesh. Furthermore, mapping tasks further apart allows greater use of the multiple paths provided by adaptive routing algorithms. This is beneficial when tasks generate multiple messages destined for the same processor. Mapping heuristics that recognize this and modify mappings to benefit from these situations are likely to show greater improvement than existing heuristics. In addition, adaptive routing algorithms are frequently imbalanced. Thus, the degree of adaptiveness is not distributed uniformly among pairs of processors. Better mappings could be produced by incorporating information about imbalance into the mapping heuristic.

To a certain degree, mapping heuristics that reduce contention are prevented from achieving good performance unless the initial clustering is suitable. The Dominant Sequence Clustering heuristic that we have used in the simulations does not consider contention. (Since the clustering assumes a completely connected architecture, contention can occur only when multiple messages are sent to the same destination.) The merging of clusters should also consider the contention aspects and more attention should be given to reducing message traffic, especially multiple messages between the same pair of clusters, than load balancing.

7.2.7 Validity of Analytical Measures of Adaptiveness

We have used analytical measures of adaptiveness, specifically the degree of adaptiveness, to argue that our routing algorithms are better than previously proposed routing algorithms. There are reasons to believe that, for our routing algorithms, this argument is valid. First, the router complexity and network cycle times are the same for our routing algorithms and previously proposed fully adaptive routing algorithms. Second, some simulation work [38], including the simulation results presented in Chapter VI, supports this belief. Finally, intuition suggests that increasing adaptiveness without increasing router complexity should lead to better performance. An increase in degree of adaptiveness does not necessary imply an increase in the number of independent paths, however, so other analytical measures of adaptiveness may be required. In cases where an increase in adaptiveness requires an increase in router complexity, however, the trade-off is even less clear. Simulation studies, particularly with message traffic from real applications, should provide a better understanding of the relationship between the degree of adaptiveness and the actual performance of routing algorithms.

7.3 Future Relevance

With the rapid changes in computer technology, any research can lose its relevance. The research in this dissertation, however, has the potential to remain applicable to future technology. The necessary and sufficient condition that has been proposed is universal. Whatever switching technique is proposed in the future, the likelihood that our methodology will remain applicable is reasonably high. As long as deadlock is avoided by routing restrictions, rather than, using switching techniques that are inherently deadlock-free, our technique is applicable. Placing limited buffer space on the routers and using a combination of virtual cut-through and wormhole routing, as on the IBM SP2, seems to be the leading candidate for future interconnection networks. The need to provide low message latency on a large-scale multiprocessor seems to require the use of some variant of virtual cut-through or wormhole routing.

Since wormhole routing is more prone to deadlock than packet switching or virtual cut-through, the routing algorithm for meshes that we have proposed can be used with these switching techniques as well. Although the optimality proofs do not hold for other switching techniques, the high degree of adaptiveness should be useful.

With the current trends in architectural and technological improvements, message start-up times will continue to decrease and processor speeds will continue to increase. Naturally, channel contention in the interconnection network will also increase. Further, with the trend toward massively parallel processing, the parallel program will be partitioned into increasingly smaller components. This small granularity makes the performance more dependent on message latency, because more messages are likely to be generated. Channel contention will thus become even more of an issue in future multiprocessors.

Bibliography

[1] I. Ahmad and Y.-K. Kwok. A New Approach to Scheduling Parallel Programs Using Task Duplication. In *International Conference on Parallel Processing*, volume II, pages 47–51, 1994.

[2] P. Berman, L. Gravano, G. Pifarré, and J. Sanz. Adaptive Deadlock- and Livelock-Free Routing With All Minimal Paths in Torus Networks. In 4^{th} *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–12, 1992.

[3] S. H. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, C-30(3):207–214, March 1981.

[4] R. V. Bopanna and S. Chalasani. New Wormhole Routing Algorithms for Multicomputers. In *International Parallel Processing Symposium*, pages 419–423, 1993.

[5] Y. M. Boura and C. R. Das. A Class of Partially Adaptive Routing Algorithms for n_dimensional Meshes. In *International Conference on Parallel Processing*, volume III, pages 175–182, 1993.

[6] Y. M. Boura and C. R. Das. Efficient fully adaptive wormhole routing in n-dimensional Meshes. In *International Conference on Distributed Computing Systems*, pages 589–596, 1994.

[7] V. Chaudhary and J. K. Aggarwal. A Generalized Scheme for Mapping Parallel Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):328–346, March 1993.

[8] A. A. Chien. A Cost and Speed Model for k-ary n-cube Wormhole Routers. In *Hot Interconnects '93*, August 1993.

[9] A. A. Chien and J. H. Kim. Planar-Adaptive Routing: Low-cost Adaptive Networks for Multiprocessors. In 19^{th} *Annual International Symposium on Computer Architecture*, pages 268–277, 1992.

[10] S. Chittor and R. Enbody. Predicting the Effect of Mapping on the Communication Performance of Large Multicomputers. In *International Conference on Parallel Processing*, volume II, pages 1–4, 1991.

[11] R. Cypher and L. Gravano. Adaptive, Deadlock-Free Packet Routing in Torus Networks with Minimal Storage. In *International Conference on Parallel Processing*, volume III, pages 204–211, 1992.

[12] R. Cypher and L. Gravano. Requirements for Deadlock-Free, Adaptive Packet Routing. *SIAM Journal on Computing*, 23(6):1266–1274, December 1994.

[13] W. J. Dally. Fine-Grain Message-Passing Concurrent Computers. In *Proceedings of the Third Conference on Hypercube Concurrent Computers*, volume 1, pages 2–12, 1988.

[14] W. J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transactions on Computers*, C-39(6):775–785, June 1990.

[15] W. J. Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.

[16] W. J. Dally and H. Aoki. Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):466–475, April 1993.

[17] 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.

[18] S. Darbha and D. P. Agrawal. A Task Duplication Based Optimal Scheduling Algorithm For Variable Execution Time Tasks. In *International Conference on Parallel Processing*, volume II, pages 52–56, 1994.

[19] S. Darbha and D. P. Agrawal. SDBS: A Task Duplication Based Optimal Scheduling Algorithm. In *Scalable High-Performance Computing Conference*, pages 756–763, 1994.

[20] V. A. Dixit-Radiya and D. K. Panda. Task Assignment on Distributed-Memory Systems with Adaptive Wormhole Routing. In *Symposium on Parallel and Distributed Processing*, pages 674–681, 1993.

[21] V. A. Dixit-Radiya and D. K. Panda. Clustering and Intra-Processor Scheduling for Explicitly-Parallel Programs on Distributed-Memory Systems. In *International Parallel Processing Symposium*, pages 609–615, 1994.

[22] J. T. Draper and J. Ghosh. Multipath E-Cube Algorithms (MECA) for Adaptive Wormhole Routing and Broadcasting in k-ary n-cubes. In *International Parallel Processing Symposium*, pages 407–410, 1992.

[23] J. Duato. On the Design of Deadlock-Free Adaptive Routing Algorithms for Multicomputers: Design Methodologies. In *Parallel Architectures and Languages Europe 91*, volume I, pages 390–405, 1991.

[24] J. Duato. A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Wormhole Networks. Technical report, Universidad Politecnica de Valencia, 1993.

[25] J. Duato. A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1320–1331, December 1993.

[26] J. Duato. A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Cut-Through and Store-and-Forward Networks. *IEEE Transactions on Parallel and Distributed Systems*, 1994. Submitted for publication.

[27] J. Duato. A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Wormhole Networks. In *International Conference on Parallel Processing*, volume I, pages 142–149, 1994.

[28] J. Duato. Private Communication, May 1994.

[29] J. Duato and P. López. Performance Evaluation of Adaptive Routing Algorithms for k-ary n-cubes. *Lecture Notes in Computer Science*, 853:45–59, 1994.

[30] H. El-Rewini and T. G. Lewis. Scheduling Parallel Program Tasks onto Arbitrary Target Machines. *Journal of Parallel and Distributed Computing*, 9:138–153, June 1990.

[31] S. Felperin, L. Gravano, G. Pifarré, and J. Sanz. Fully-Adaptive Routing: Packet Switching Performance and Wormhole Algorithms. In *Supercomputing '91*, pages 654–662, 1991.

[32] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, New York, NY, 1979.

[33] P. T. Gaughan and S. Yalamanchili. Pipelined Circuit-Switching: A Fault-Tolerant Variant of Wormhole Routing. In *Symposium on Parallel and Distributed Processing*, pages 148–155, 1992.

[34] P. T. Gaughan and S. Yalamanchili. Adaptive Routing Protocols for Hypercube Interconnection Networks. *IEEE Computer*, 26(5):12–23, May 1993.

[35] 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.

[36] K. Ghose and N. Mehdiratta. A Universal Approach for Task Scheduling for Distributed Memory Multiprocessors. In *Scalable High-Performance Computing Conference*, pages 577–584, 1994.

[37] C. Glass and L. M. Ni. Adaptive Routing in Mesh-Connected Networks. In 12^{th} *International Conference on Distributed Computing Systems*, pages 12–19, 1992.

[38] 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.

[39] C. Glass and L. M. Ni. The Turn Model for Adaptive Routing. In 19^{th} *Annual International Symposium on Computer Architecture*, pages 278–287, 1992.

[40] C. Glass and L. M. Ni. Fault-Tolerant Wormhole Routing in Meshes. In 23^{rd} *Annual International Symposium on Fault-Tolerant Computing*, pages 240–249, 1993.

[41] C. Glass and L. M. Ni. The Turn Model for Adaptive Routing. *Journal of the Association for Computing Machinery*, 41(5):874–902, September 1994.

[42] I. S. Gopal. Prevention of Store-and-Forward Deadlock in Computer Networks. *IEEE Transactions on Communications*, COM-33(12):1258–1264, December 1985.

[43] L. Gravano, G. Pifarré, G. Denicolay, and J. Sanz. Adaptive Deadlock-free Worm-hole Routing in Hypercubes. In *International Parallel Processing Symposium*, pages 512–515, 1992.

[44] B. Greenblatt and C. J. Linn. Branch and Bound Style Algorithms for Scheduling Communicating Tasks in a Distributed System. In *Compcon '87*, pages 12–17, 1987.

[45] K. D. Günther. Prevention of Deadlocks in Packet-Switched Data Transport Systems. *IEEE Transactions on Communications*, COM-29(4):512–524, April 1981.

[46] D. N. Jayasimha. Partially Shared Variables and Hierarchical Shared Memory Multiprocessor Architectures. In 11^{th} *IEEE International Phoenix Conference on Computers and Communications*, pages 63–71, 1992.

[47] D. N. Jayasimha, J. May, K. Patel, R. Rao, and L. Schwiebert. *A Simulation Environment for Multiprocessor Architectures*, 1995. Working Document.

[48] C. R. Jesshope, P. R. Miller, and J. T. Yantchev. High Performance Communications in Processor Networks. In 16^{th} *Annual International Symposium on Computer Architecture*, pages 150–157, 1989.

[49] H. F. Jordan. Shared Versus Distributed Memory Multiprocessors. Technical Report 91-7, Institute for Compter Applications in Science and Engineering (ICASE), January 1991.

[50] H. Jung, L. Kirousis, and P. Spirakis. Lower Bounds and Efficient Algorithms for Multiprocessor Scheduling of DAGs with Communication Delays. In 1^{st} *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 254–264, 1989.

[51] K. Kalpakis and Y. Yesha. Optimal within a Constant Schedules for Forest Dags on Parallel Architectures. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 158–163, 1993.

[52] P. Kermani and L. Kleinrock. Virtual Cut-Through: A New Computer Communication Switching Technique. *Computer Networks*, 3(4):267–286, September 1979.

[53] A. A. Khan, C. L. McCreary, and M. S. Jones. A Comparison of Multiprocessor Scheduling Heuristics. In *International Conference on Parallel Processing*, volume II, pages 243–250, 1994.

[54] J. H. Kim, Z. Liu, and A. A. Chien. Compressionless Routing: A Framework for Adaptive and Fault-tolerant Routing. In 21^{st} *Annual International Symposium on Computer Architecture*, pages 289–300, 1994.

[55] S. J. Kim and J. C. Browne. A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures. In *International Conference on Parallel Processing*, volume III, pages 1–8, 1988.

[56] S. Konstantinidou and L. Snyder. Chaos router: architecture and performance. In 18^{th} *Annual International Symposium on Computer Architecture*, pages 212–221, 1991.

[57] S. Kon'ya and T. Satoh. Task Scheduling on a Hypercube with Link Contentions. In *International Parallel Processing Symposium*, pages 363–368, 1993.

[58] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, 5(1):23–32, January 1988.

[59] Y.-K. Kwok and I. Ahmad. A Static Scheduling Algorithm Using Dynamic Critical Path for Assigning Parallel Algorithms onto Multiprocessors. In *International Conference on Parallel Processing*, volume II, pages 155–159, 1994.

[60] S. Lee and J. K. Aggarwal. A Mapping Strategy for Parallel Processing. *IEEE Transactions on Computers*, C-36(4):433–442, April 1987.

[61] S. Lee and J. Kim. Path Selection for Communicating Tasks in a Wormhole-Routed Multicomputer. In *International Conference on Parallel Processing*, volume III, pages 172–175, 1994.

[62] Q. Li. Minimum Deadlock-Free Message Routing Restrictions in Binary Hypercubes. *Journal of Parallel and Distributed Computing*, 15(2):153–159, June 1992.

[63] X. Lin, A.-H. Esfahanian, P. K. McKinley, and A. Burago. Adaptive Wormhole Routing in Hypercube Multicomputers. In *Symposium on Parallel and Distributed Processing*, pages 72–79, 1993.

[64] X. Lin, P. K. McKinley, and L. M. Ni. The Message Flow Model for Routing in Wormhole-Routed Networks. In *International Conference on Parallel Processing*, volume I, pages 294–297, 1993.

[65] X. Lin and L. Ni. Deadlock-Free Multicast Wormhole Routing in Multicomputer Networks. In 18^{th} *Annual International Symposium on Computer Architecture*, pages 116–125, 1991.

[66] D. H. Linder and J. C. Harden. An Adaptive and Fault Tolerant Wormhole Routing Strategy for k-ary n-cubes. *IEEE Transactions on Computers*, 40(1):2–12, January 1991.

[67] Z. Liu and A. A. Chien. Hierarchical Adaptive Routing. In *Symposium on Parallel and Distributed Processing*, pages 688–695, 1994.

[68] Z. Liu, J. Duato, and L.-E. Thorelli. Grouping Virtual Channels for Deadlock-Free Adaptive Wormhole Routing. In *Parallel Architectures and Languages Europe 93*, pages 254–265, 1993.

[69] V. M. Lo *et al.* OREGAMI: Tools for Mapping Parallel Computations to Parallel Architectures. *International Journal of Parallel Programming*, 20(3):237–270, June 1991.

[70] P. López and J. Duato. Deadlock-Free Adaptive Routing Algorithms for the 3D-Torus: Limitations and Solutions. In *Parallel Architectures and Languages Europe 93*, pages 684–687, 1993.

[71] W. Löwe and W. Zimmermann. On Finding Optimal Clusterings of Task Graphs. In *First Aizu International Symposium on Parallel Algorithms and Architecture Synthesis (pAs 95)*, pages 241–247, 1995.

[72] P. Markenscoff and Y. Y. Li. Scheduling a Computational Dag on a Parallel System with Communication Delays and Replication of Node Execution. In *International Parallel Processing Symposium*, pages 113–117, 1993.

[73] J. May, D. N. Jayasimha, and K. Patel. Comparison of Multiplexing Schemes for Wormhole-Routed Distributed Memory Multiprocessors. In 1^{st} *International Workshop on Parallel Processing*, pages 211–215, 1994.

[74] C. McCreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32(9):1073–1078, September 1989.

[75] N. Mehdiratta and K. Ghose. A Bottom-Up Approach to Task Scheduling on Distributed Memory Multiprocessors. In *International Conference on Parallel Processing*, volume II, pages 151–154, 1994.

[76] J. Y. Ngai and C. L. Seitz. A Framework for Adaptive Routing in Multicomputer Networks. In 1^{st} *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–9, 1989.

[77] L. M. Ni and P. K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, 26(2):62–76, February 1993.

[78] D. K. Panda and V. A. Dixit-Radiya. Message-Ordering for Wormhole-Routed Multiport Systems with Link Contention and Routing Adaptivity. In *Scalable High-Performance Computing Conference*, pages 191–198, 1994.

[79] C. H. Papadimitriou and M. Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. *SIAM Journal on Computing*, 19(2):322–328, April 1990.

[80] 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.

[81] G. F. Pfister and V. A. Norton. "Hot-Spot" Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[82] G. D. Pifarré, L. Gravano, S. A. Felperin, and J. L. C. Sanz. Fully-Adaptive Minimal Deadlock-Free Packet Routing in Hypercubes, Meshes and Other Networks. In 3^{rd} *Annual Symposium on Parallel Algorithms and Architectures*, pages 278–289, 1991.

[83] G. D. Pifarré, L. Gravano, S. A. Felperin, and J. L. C. Sanz. Fully Adaptive Minimal Deadlock-Free Packet Routing in Hypercubes, Meshes, and Other Networks: Algorithms and Simulations. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):247–263, March 1994.

[84] F. P. Preparata and J. E. Vuillemin. The cube-connected-cycles: A Versatile Network for Parallel Computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[85] M. R. Samatham and D. K. Pradhan. The De Bruijn Multiprocessor Network: A Versatile Parallel Processing and Sorting Network for VLSI. *IEEE Transactions on Computers*, 38(4):567–581, April 1989.

[86] V. Sarkar and J. Hennessy. Compile-time Partitioning and Scheduling of Parallel Programs. In *ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 17–26, 1986.

[87] H. Schwetman. CSIM: A C-Based, Process-Oriented Simulation Language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, 1986.

[88] L. Schwiebert and D. N. Jayasimha. A Necessary and Sufficient Condition for Deadlock-Free Wormhole Routing. *Journal of Parallel and Distributed Computing*. To appear.

[89] L. Schwiebert and D. N. Jayasimha. Mapping to Reduce Contention in Multiprocessor Architectures. In *International Parallel Processing Symposium*, pages 248–253, 1993.

[90] L. Schwiebert and D. N. Jayasimha. Optimal Fully Adaptive Wormhole Routing for Meshes. In *Supercomputing '93*, pages 782–791, 1993. An extended version of this paper will appear in the Journal of Parallel and Distributed Computing.

[91] L. Schwiebert and D. N. Jayasimha. A Universal Proof Technique for Deadlock-Free Routing in Interconnection Networks. In 7^{th} *Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1995.

[92] X. Shen and Y. S. Cheung. The B-&-E Model for Adaptive Wormhole Routing. In *Symposium on Parallel and Distributed Processing*, pages 170–173, 1993.

[93] G. C. Sih and E. A. Lee. Scheduling to Account for Interprocessor Communication within Interconnection-Constrained Processor Networks. In *International Conference on Parallel Processing*, volume I, pages 9–16, 1990.

[94] G. C. Sih and E. A. Lee. A Compile-Time Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–186, February 1993.

[95] C.-C. Su and K. G. Shin. Adaptive Deadlock-Free Routing in Multicomputers Using Only One Extra Virtual Channel. In *International Conference on Parallel Processing*, volume I, pages 227–231, 1993.

[96] H. Sullivan and T. R. Bashkow. A Large Scale, Homogeneous Fully Distributed Parallel Machine. In 4^{th} *Annual Symposium on Computer Architecture*, volume 5, pages 105–124, 1977.

[97] R. Thurimella and Y. Yesha. A Scheduling Principle for Precedence Graphs with Communication Delay. In *International Conference on Parallel Processing*, volume III, pages 229–236, 1992.

[98] S. Toueg and K. Steiglitz. Some Complexity Results in the Design of Deadlock-Free Packet Switching Networks. *SIAM Journal on Computing*, 10(4):702–712, November 1981.

[99] J. Upadhyay, V. Varavithya, and P. Mohapatra. Efficient and Balanced Adaptive Routing in Two-Dimensional Meshes. In *First International Symposium on High-Performance Computer Architecture*, pages 112–121, 1995.

[100] M. Wang, B. Lee, B. Shirazi, and A. R. Hurson. Accurate Communication Cost Estimation in Static Task Scheduling. In *Hawaii International Conference on System Sciences*, volume 1, pages 10–16, 1991.

[101] C. S. Yang and Y. M. Tsai. Fault-Tolerant and Deadlock-Free Wormhole Routing in the Hypercube Network. In *International Conference on Parallel and Distributed Systems*, pages 9–16, 1992.

[102] 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.

[103] T. Yang and A. Gerasoulis. A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors. In *ACM International Conference on Supercomputing*, pages 633–642, 1991.

[104] 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.

[105] 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.