ABSTRACT

Mapping and Scheduling of Concurrent Communication Traffic in Multicomputer Networks

by
Bingrung Tsai

Chair: Kang G. Shin

In a multicomputer network, each task is decomposed into modules which are then executed by a number of processing nodes. During the execution, bursty concurrent communication traffic among processor nodes can lead to network congestion and become a bottleneck. Our goal is to improve the system communication performance in such a situation using on-line traffic flow-control mechanisms, and develop task mapping strategies with sufficient task behavior information prior to the execution.

Flow-control mechanisms include message routing algorithms, message scheduling policies, and in the case of virtual channel networks, time-division multiplexing methods. For on-line applications, we propose low-complexity distributed mechanisms to be implemented on each node. Using simulations, we show that effective mechanisms can improve network performance significantly. Without proper flow-control mechanisms, adding communication resources such as links and buffers can actually degrade the network performance. Also, under heavy concurrent traffic, a low-complexity routing algorithm, when combined with a good message scheduling policy, can outperform a much more complex routing algorithm and achieve near-optimal performance.

Before their execution, if the task communication behavior can be predicted (with a certain degree of accuracy), task modules can be mapped to processor nodes strategically to exploit their communication locality and thus improve the run-time performance. However, due to the complex interactions among these modules, the performance objective can be very difficult to formulate, making direct optimization impossible. Thus, we propose simple cost functions so that when mappings are optimized with respect to these functions, the actual performance objectives are also optimized. For this purpose, various heuristic algorithms are developed and tested through analytical modeling and simulations. Several cost functions are studied and a few of them found to be very effective when used with appropriate algorithms. For binary hypercubes, we also extend our study to the problem of mapping concurrently communicating subcubes. We show that the optimization process can be simplified and existing algorithms can be easily adapted for this application.

Mapping and Scheduling of Concurrent Communication Traffic in Multicomputer Networks

by

Bingrung Tsai

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1994

Doctoral Committee:
Professor Kang G. Shin, Chair
Associate Professor Kevin J. Compton
Associate Research Scientist Chinya V. Ravishankar
Professor Thomas F. Storer
Professor Quentin F. Stout

To my parents

ACKNOWLEDGEMENTS

I'd like to express my deepest gratitude to my advisor Prof. Kang G. Shin. The constant guidance, timely encouragement and unwavering support he provided me to pursue this research has been priceless during my entire graduate study.

Many thanks are extended to the members of my committee for their constructive criticism and time spent in reviewing this dissertation. Also, I am obliged to the Office of Naval Research for their financial support.

I am forever grateful to my parents for the support and encouragement they have given me that made my graduate study possible. Finally, a very special thank goes to my fiancee Szu-ping (Grace), with whom I feel very fortunate to be able to share my life, for her encouragement and confidence in me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure

CHAPTER 1

INTRODUCTION

1.1 Motivation

The computing speed of a single computer has been greatly improved over the last two decades. However, there are intrinsic limitations in the computing capability of a single computer. Computer scientists have always been exploring the possibility to make multiple computers to work in parallel to achieve better performance.

Due mainly to the availability of inexpensive, high-performance microprocessors and new networking technology, it has recently become very attractive to link together many autonomous computers, called *computing nodes*, to build a multicomputer system for better performance. In a multicomputer, a computational task is divided into smaller modules, and each module is assigned to a computing node. To coordinate these nodes and ensure that they cooperatively execute a common task, there needs to be a communication network so that any node can send messages to any other node. In most situations, these computing nodes do not work independently. One node may need to wait for messages from some other nodes to continue its work. Therefore, if communications are not done efficiently, the overall computing performance of a multicomputer can be seriously degraded. This has been recognized as a major bottleneck in the performance of existing multicomputers [25, 66, 42].

The communication issue becomes even more of a concern if there are a large number of nodes trying to send messages to other nodes (almost) *concurrently*. This occurs quite often if many nodes need to exchange their partial results or synchronize their executions. Consequently, a congestion, which is similar to a traffic jam, in the network occurs and messages may need a long time to get through. Worse yet, before these messages reach their destinations, many nodes cannot continue their computation and thus, the overall performance of the multicomputer suffers. It is therefore important to improve the network's ability to handle this kind of concurrent traffic arrivals.

1.2 Preliminaries and Previous Research

1.2.1 Multicomputer Networks

Our research focuses mainly on multicomputer systems with a large number of processor nodes. In these systems, instead of global memory, each processor has its own local memory, so each processor does not have immediate access to all of the available data. If a required datum is not in a processor's local memory, it must request and receive the datum from another processor whose memory stores the datum. The processors are connected via a *network* for such communication. We will only deal with *direct* networks, in which processors communicate with one another by using dedicated communication links. In contrast, in an *indirect* network, processors communicate through a series of intermediate devices, and is often referred to as a *multistage interconnection network*.

In our study, we will focus on the network topology family called k-ary n-cubes, and particularly, binary hypercubes where $k = 2$, and meshes where $n = 2$. A binary 4-cube is shown in Fig 1.1, and a 4-ary 2-cubes shown in Fig. 1.2. An example of the existing multicomputers using the binary hypercube topology is the Cosmic Cube [96]. On the other hand, the J-Machine [84] uses a mesh-like network with $n = 3$.

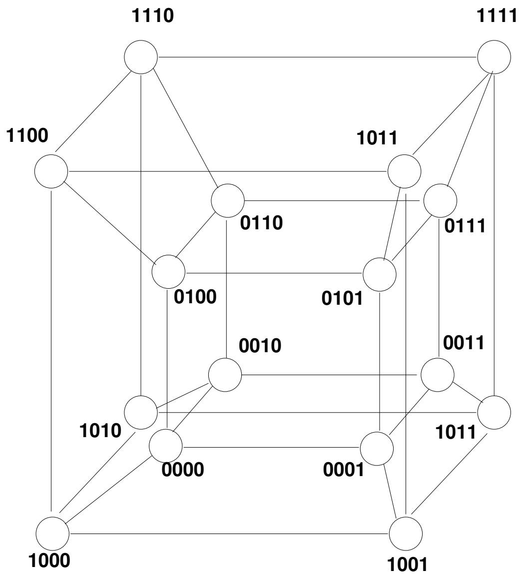


Figure 1.1: A binary 4-cube.

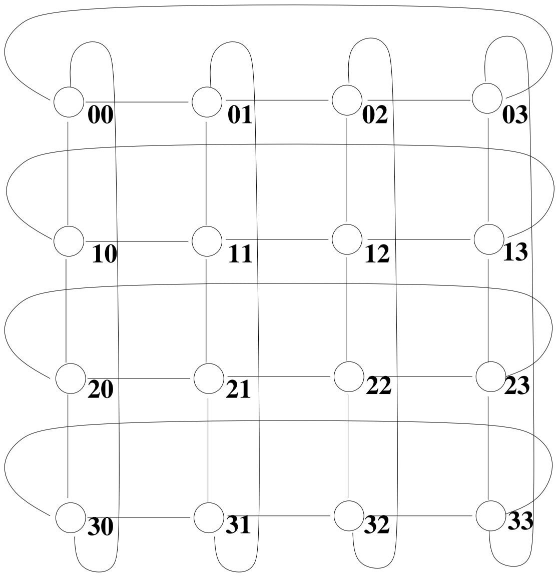


Figure 1.2: A 4-ary 2-cube.

1.2.2 Communication Mechanisms

Switching Methods

There are four basic switching methods for interprocessor communication in multicomputers: message switching, circuit switching, virtual cut-through, and wormhole switching.

Message switching behaves in a store-and-forward manner, analogous to the usual mail service. This method is generally advantageous when messages are short and/or infrequent. Minimum message latency is proportional to the product of the number of hops and the message length.

Circuit switching behaves like telephone systems. A path from the source to the destination is initially established and the circuit is held until the entire message is transmitted. This method is generally most effective when messages are long and/or infrequent. Minimum message latency is proportional to the sum of the message length and some constant (> 1) multiple of the path length.

Wormhole switching attempts to combine the benefits of message switching and circuit switching. The message is broken into small parcels called *flow control digits*, or *flits*, which are pipelined through the network. As soon as enough routing information becomes available at an intermediate node and a corresponding output link is free, forwarding of flits begins even before the entire message is received. If no out-going link is free, flits are buffered at intermediate nodes. If the buffers are not large enough, the message will be buffered across several intermediate nodes, thus holding the links between them. Minimum latency is proportional to the sum of the message length and the path length.

Virtual cut-through is a special case of wormhole switching in which the buffers at any intermediate node are large enough to hold the entire message. There is no link held when messages are blocked.

Virtual Channels

A virtual channel is a logical entity associated with a physical link used to distinguish multiple data streams traversing the same physical channel. Messages in virtual channels are time-multiplexed over the physical channel by the underlying flow-control mechanism.

Virtual channels are used for deadlock avoidance by imposing routing restrictions [24]. They can also increase network throughput by reducing physical link idle time [27], especially in wormhole-switched networks. If virtual channels are not used in wormhole-switched networks, when a routing header blocks, all physical links in the path are idle until the header can advance. When virtual channels are used, other messages can use these physical links while the blocked header waits.

1.2.3 Traffic Models

In this dissertation, we will mostly discuss the network *transient* behaviors under heavy communication traffic arrivals. A group of messages which are sent through the network in a small time span is called a *communication mission*, or *mission* for short. The time required for a network to complete a mission is called *makespan*. Once the communication mission is in progress, no additional messages are allowed into the network. This was called the *non-renewal context* in [45] and had been introduced in [111, 110] for synchronous communications. This traffic model was also used in [103, 53, 54, 61, 113].

The opposing context to the non-renewal context is called the *renewal context* [45], in which messages are generated and allowed into the network continually and asynchronously. Analysis of the renewal context is mostly applied to situations in which the system has reached a steady-state [82, 25, 26, 27, 42, 43, 1].

1.2.4 Traffic Flow Control

In a multicomputer network, traffic flow-control mechanisms include message routing algorithms, message scheduling policies, and in the case of virtual-channel networks, time-division multiplexing methods. Flow-control mechanisms have little to do with the switching method used in a network. In general, any flow-control mechanism can be used with all four switching methods mentioned above.

Routing

Routing is the process of determining the path for a message to traverse in order to reach its destination. Routing decision can be made using a *centralized* algorithm, which globally coordinates the paths for all messages, or using a *distributed* algorithm, which makes routing decisions locally on each processor node.

A centralized algorithm is useful only if the traffic pattern of a set of messages is known *a priori*, or the network load condition is stored in a *routing table* whose contents are updated regularly. When using a centralized routing algorithm, message paths can be chosen to minimize contention so that the performance can often be optimal or near-optimal [103, 53, 54, 16]. For on-line applications, the information required to perform centralized routing is not easily obtained, and hence, a distributed algorithm is more practical.

Distributed algorithms can be *oblivious*, in which a fixed path is chosen for a given pair of source and destination nodes regardless of the network traffic condition. This type of algorithms make poor use of network bandwidth, thus blocking messages even when alternative paths are available. They are also particularly susceptible to component failures. One of the most commonly used oblivious routing algorithms for k-ary n-cubes is the *e-cube* algorithm. Routing proceeds traversing a fixed sequence of dimensions. When the difference in the address of the current node and that of the destination node is 0 in dimension i, the message is said to have *traversed* dimension i. Once all dimensions are successfully traversed, the message is at the destination node.

In contrast, *adaptive* algorithms can use alternative paths between communicating processor nodes, making more efficient use of network bandwidth and providing resilience to failures. When an adaptive algorithm is used in a k-ary n-cube, the traversal of dimensions is not fixed in a particular sequence. Adaptive routing algorithms have received considerable attention from researchers [18, 32, 44, 5, 8, 11, 12, 20, 22, 23, 37, 36, 49, 48, 52, 56, 60, 83, 90], and can be further classified as *fully* or *partially* adaptive. A fully adaptive algorithm can use all possible paths between the source and destination, and requires certain mechanism to prevent a livelock. For example, in [18], the algorithm adds an extra field to the message to record its routing "history". A partially adaptive algorithm explores a subset of possible paths, e.g., the shortest ones only, and is generally of lower complexity than a fully adaptive one.

Although most simulation studies have shown adaptive algorithms to have superior performance over oblivious algorithms, the practicality of building an adaptive network router has been questioned due to the router's complexity. Also, adaptive routers must make more complex decisions and maybe slower, hence offsetting their performance advantages over oblivious routers. However, some of the most recent high-performance routers [10, 31] are able to support less complex adaptive routing without incurring significant overhead over oblivious routing. This demonstrates the feasibility of low-complexity adaptive routing.

Message Scheduling

Message scheduling is the process of determining which message is allowed to access communication resource in case of contention. In effect, a message-scheduling policy orders messages in the queues. Message scheduling policies can also be either centralized or distributed.

In centralized policies, full knowledge of traffic patterns and message arrival times is necessary. The problem is treated as a classical scheduling problem [41, 81] for general traffic patterns, or as an integral part of routing algorithms [103, 53, 54, 61] for some specific traffic patterns.

In distributed policies, only the information already contained in a message is available to a scheduler on a processor node. Research on distributed message scheduling has been mostly in the computer network literature [112, 116, 15]. On a multicomputer, most low-complexity scheduling policies can be implemented with small modifications to the architecture of message queuing mechanisms [27, 28].

Time-Division Multiplexing

In a network with virtual channels, time-division multiplexing methods determines the way messages in different virtual channels are time-multiplexed over a physical channel. Usually, messages are partitioned into flits and multiplexing methods determines the order in which the flits from different messages access the physical channel.

Due to the complex nature of multiplexing, multiplexing methods are rarely centralized. Also, since flit multiplexing is performed on a flit-by-flit basis, a feasible multiplexing method must be of very low complexity and incur extremely low overhead. Time-division multiplexing has been discussed mostly in the literature related to the ATM networks [88]. On a multicomputer, several low-complexity multiplexing methods have been discussed in [43]. A more complex, age-based method was used in [27].

1.2.5 Task Mapping

Before executing a task, if there is sufficient knowledge of the behavior of the task, then this task can be mapped onto the multicomputer to improve their run-time performance. In a multicomputer system, a task is usually decomposed into a set of cooperating task modules which are then assigned to a set of processors. Therefore, a task-mapping problem is essentially a module-mapping problem.

Models and Formulations

As shown in the survey by Norman and Thanisch [85], the problem of task mapping has been addressed by numerous researchers using a wide range of models. They can be generally categorized into 5 basic types [85].

- Model 1: No precedence relationship among modules
- Model 2: Precedence relationship but with no communication cost
- Model 3: Precedence relationship and with communication delay
- Model 4: Both communication costs and computation costs
- Model 5: Communication costs only

Model 1 is the simplest and most computationally tractable of all. Each of the modules are executed sequentially on a single processor and there is no inter-module communication. It also often referred to as *event parallel* [91] and related to the bin-packing problem [41]. Theoretical results and performance bounds regarding this model have been derived in [14, 47, 63]. Some researchers [65, 40] have also extended Model 1 to allow certain constraints such as on the locations of modules that can be mapped to.

In Model 2, the modules communicate results to other modules on their completion of execution, and the structure of the computation is represented as a directed graph in which a directed arc connects a pair of modules if and only if the module at the source of the arc requires the results of computation from the module at the destination of the arc. However, it is assumed that there is no communication cost. Since there is no communication delay, processors are never idle waiting for messages to transmit, but can be idle waiting for a precedence relation to be satisfied. Complexity results related to this model were presented in [40, 51, 30]. Algorithms or approximate algorithms for this model include those discussed in [19, 58, 67, 39].

Model 3 is an elaboration of Model 2 by adding a weight on each arc of the task graph to indicate the cost of an instance of communication. Complexity results of this model were reported in [95, 87, 64]. Approximation algorithms were discussed in [87, 105, 73, 59], and some heuristics can be found in [115, 6].

Models 4 and 5 differ from Models 2 and 3 in that modules are mostly represented by undirected graphs. They are often used for modeling computations at a coarser granularity where there are predictable communication patterns.

Model 4 can be typified by Stone's [101] model, in which there is a computation cost associated with each module and a communication cost associated with each message. The total cost of a mapping is the sum of all the computation costs and communication costs. Results based on Stone's model were also presented in [102, 9, 106, 94, 50].

In cases where computation and communication costs are not incurred sequentially, e.g., in parallel rather than serial programs, Stone's model is not easily applicable since communication costs and computation costs are no longer additive. It may be more practical to find minimum communication cost mappings without considering computation costs, and hence, Model 5 is used. Models of this category often include considerations of some underlying processor architecture. The problem becomes that of mapping an undirected graph of modules into an undirected graph of processors so as to minimize the communication overhead. The overhead is typically defined as some mismatch between the edges of the processor graph and the edges of the task graph. As Fox [38] observed in a survey of parallel applications, 76% of real parallel programs can be considered as synchronous or loosely synchronous. The programs would be executed on a multicomputer by regular data decomposition with one module per processor and approximately an equal amount of computation per module. Also, as a result of global or pairwise synchronizations between modules, the execution would proceed mostly in a lock-step manner. Therefore, the problem is reduced to mapping modules to minimize communication overhead. Results based on this type of models include those presented in [7, 9, 62, 75, 74].

Other work on task mapping that uses models not easily classified includes [57, 78, 79, 21, 34]. A model which considers the tradeoff between processor and memory allocation is used in [89].

Uncertainty in Task Behavior

Most work in the task mapping literature has assumed the availability of an exact task graph. However, most compilers are not able to generate such a task graph since the task graph itself may even be underivable before the program runs. However, the effectiveness of mapping algorithms may not be guaranteed in the presence of uncertain information on task behavior. Only a few researchers have addressed this issue. In [80], a model was used to allow probabilistic branching, whereas in [100] the worst-case performance of several heuristic approaches were analyzed.

1.3 Research Objective and General Methodology

Our goal is to optimize the performance of multicomputers by improving the performance of exchanging messages among computing nodes. Especially, we will focus on situations where there are a large number of messages sent concurrently among nodes. This goal is achieved by exploiting communication locality through (i) on-line traffic flow-control mechanisms during the task execution, and, (ii) module mapping prior to the execution assuming the sufficient task behavior information is given.

For on-line flow control, our main focus is on low-complexity, distributed schemes which do not require any extra field in the default message format, and can be implemented on existing routers without incurring noticeable run-time overhead. We first study the effects of combining routing algorithms with message scheduling policies in a binary hypercube network with large-buffer switching methods such as (store-and-forward) message switching and virtual cut-through. In mesh networks equipped with virtual channels and wormhole switching, we study the effects of combining message-scheduling policies with time-division multiplexing methods. Performances of different combinations are evaluated by simulations.

Our task-mapping approach takes into account the underlying flow-control mechanisms in a multicomputer. We use a model which can be classified as Model 5 defined in [85] and consider only the communication performance of a task. However, unlike most previous work, the cost function itself is not our final objective. The quality of mappings is eventually determined by performance measurements through simulating module communications during task executions. A cost function is selected only if mappings optimized with respect to it can consistently improve the run-time performance. The effect of inaccurate task-behavior information is also considered. In the case of binary hypercubes, we also investigate the problem of mapping communicating subcubes, which is a generalized version of the task mapping problem. Both empirical methods and mathematical analyses are adopted in our study.

1.4 A Map of the Dissertation

The overall organization of this dissertation is shown in Fig. 1.3.

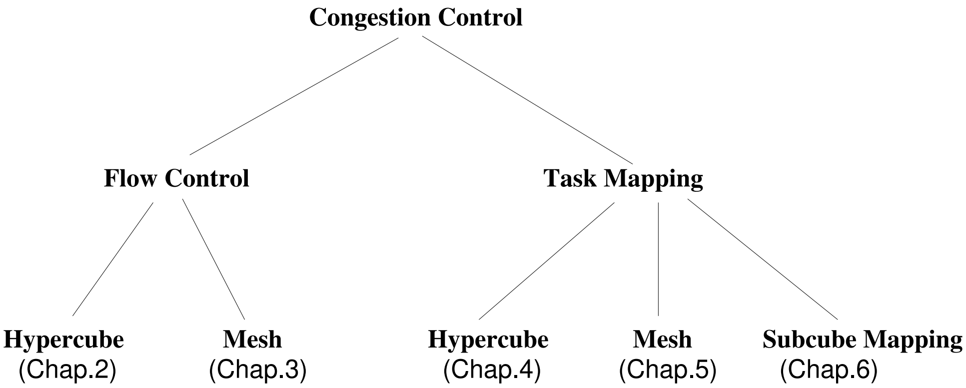


Figure 1.3: Organization of the dissertation.

In Chapter 2, we propose and evaluate low-complexity, low-overhead schemes for distributed message scheduling and routing in binary hypercube multicomputers equipped with a hardware communication adapter at each node. We comparatively evaluate the performance of different scheduling–routing combinations for several switching methods, such as message switching, circuit switching and virtual cut-through, supported by a communication adapter at each node. (Virtual channel networks with wormhole switching have very different characteristics and are thus covered in Chapter 3.) The evaluation results have indicated that a combination of carefully-chosen, low-complexity distributed message scheduling and adaptive routing can offer close to optimal performance in most situations. In case of heavy transient traffic, a low-complexity partially-adaptive routing scheme, when combined with an appropriate message-scheduling policy, can outperform a fully-adaptive routing scheme.

In Chapter 3, we propose and evaluate several low-complexity, low-overhead flow control mechanisms for a mesh multicomputer network equipped with virtual channels. In such a network, virtual-channel flow control is accomplished at three levels: message routing, message sequencing, and flit multiplexing. Under the fixed-path e-cube routing in mesh multicomputers, we evaluate the performance of several on-line message scheduling and flit multiplexing methods. In the presence of concurrent inter-node communication traffic, we found that unless proper flow-control mechanisms are employed, adding more communication resources, such as links and buffers, can actually degrade the network performance. A good message-sequencing policy combined with proper flit multiplexing is shown to improve performance by more than 30% .

In Chapters 2 and 3, we discuss on-line flow-control mechanisms. From Chapters 4 to 6, we address the issues of exploiting communication locality before the actual execution through task mapping. In Chapter 4, the problem of mapping task modules into a binary hypercube multicomputer is investigated. A concept of *indirect optimization* is introduced and a function, called *communication bandwidth*, is proposed as the optimization cost function. The mappings obtained from optimizing this function are shown to significantly improve the actual communication performance (i.e., makespan) over random mappings. We also extend our study to include the case of uncertain task behaviors resulting from link failures and adaptive message routing algorithms.

In Chapter 5, we address the task mapping problem in a virtual-channel network as defined in Chapter 3. The system under consideration is a mesh-connected computer with wormhole switching and virtual channels. In such a system, it is difficult to define and evaluate a meaningful performance objective when one has to consider many messages being transmitted concurrently. Therefore, an approximate cost function must be chosen so that when a module mapping is optimized with respect to the function, the actual performance of the mapping is also optimized. Several cost functions are tested using the simulated annealing optimization process. The mappings found through optimizing each cost function are then fed into a flit-level simulator to evaluate their actual performance. One particular cost function, $f_7 \mid f_3$, is found to be very effective. Given approximately the same amount of computing time in each case, mappings optimized with the function outperform these found through optimizing other cost functions. Also, an optimization process using this cost function can continually improve the performance as the given computing time increases.

In Chapter 6, we study the problem of mapping concurrently-communicating subcubes within a hypercube multicomputer so as to minimize inter-subcube communication traffic. The communication between subcubes is determined by a routing algorithm which in turn depends on the schemes proposed in [86, 18]. Our objective is to minimize the total inter-subcube communication bandwidth required. This function was shown in [107, 109] to be a good indication of the quality of mappings for concurrently-communicating modules.

We derive some important mathematical properties of subcube mappings. Methods are proposed to modify existing optimization algorithms for finding optimal mappings. A subset of all possible mappings, called *parallel mappings*, are found to possess some desirable properties. For some special case, optimal parallel mappings are also proved to be optimal among all mappings. We also evaluate several heuristic algorithms via simulations and show that, in most sub-optimal mappings found, parallel mappings still outperform non-parallel ones.

Chapter 7 concludes this dissertation with a summary of contributions and possible directions to extend our research.

CHAPTER 2

COMBINED ROUTING AND SCHEDULING OF CONCURRENT COMMUNICATION TRAFFIC IN HYPERCUBE MULTICOMPUTERS

2.1 Introduction

As pointed out in [25], the critical component of a multicomputer is its interconnection network. Many algorithms are communication rather than processing limited. Some fine-grain concurrent programs execute as few as ten instructions in response to a message [24]. To execute such programs efficiently, the communication network must be able to handle heavy concurrent traffic.

Message or traffic routing in multicomputer interconnection networks has received considerable attention [66, 93, 68, 42, 46, 103]. Most of the existing work has assumed inter-node communication traffic to be composed of a set of steady, independent flows. Network throughput or mean message latency over a certain period of time is often used as a performance measure. This type of performance evaluation reflects more of the "steady-state" behavior of a network. However, in the task level — where a computation task is decomposed into a set of communicating modules — intermodule traffic, and hence interprocessor communication when the modules are assigned to different processors, tends to be bursty. A large number of messages are often generated within a short span of time. Furthermore, delivering each of these messages may not be mutually independent. For instance, an algorithm may not continue its execution until the partial results are collected from, or exchanged among, the participating modules. Fig. 2.1 shows a multicomputer with four processors connected as a two-dimensional hypercube, or a Q_2. Suppose at time $T = 0$, the module assigned to processor 00 sends its partial results to all the other modules on processors 01, 10 and 11. The modules need to receive these partial results to complete their execution. Fig. 2.2 shows the space-time diagram of this set of concurrent messages in a (store-and-forward) message-switching network. Here message $00 \rightarrow 01$ is given priority over message $00 \rightarrow 11$ when both are contending for link $0*$, or the link between nodes 00 and 01. Obviously, the time when message $00 \rightarrow 11$ reaches its destination plays a major role in determining the overall completion time of the task.

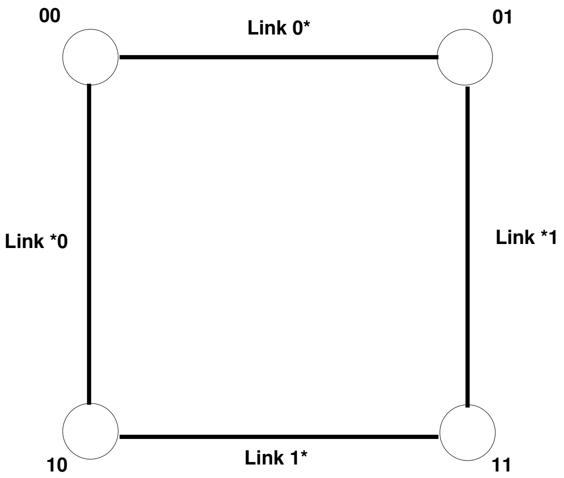


Figure 2.1: An example Q_2.

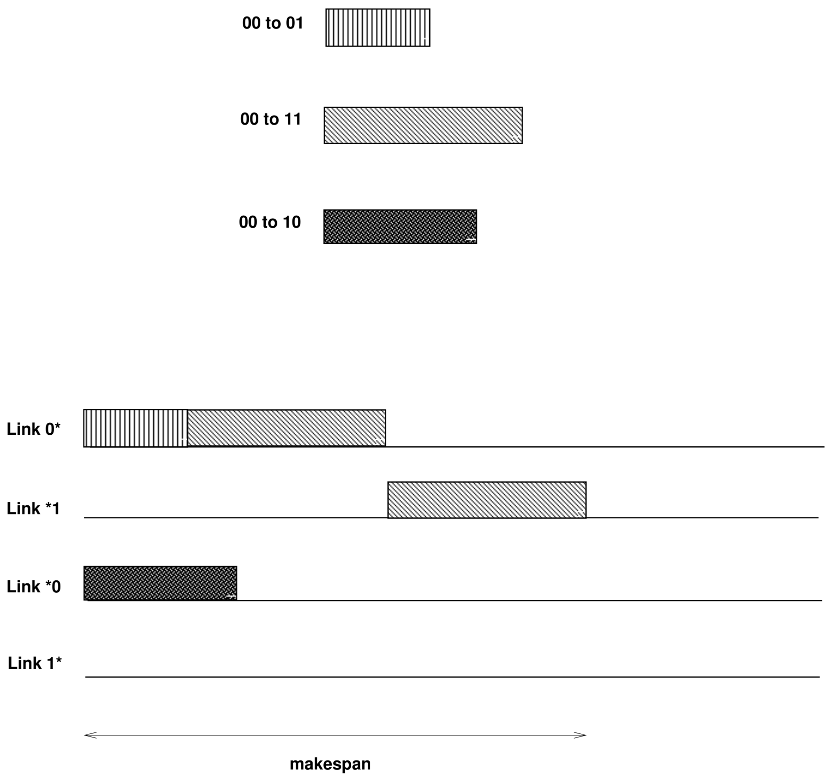


Figure 2.2: An example demonstrating the definition of makespan.

We define a *communication mission*, or *mission* for short, to be a set of messages to be exchanged among the nodes such that the completion of delivery of these messages as a whole is crucial to the completion of the whole task. The *makespan* of a mission denotes the time span since the arrival[1] of the first message until the last message reaches its destination. (A formal definition of makespan will be given in Section 2.2.) Note that in the above example, if message $00 \rightarrow 11$ is given priority over message $00 \rightarrow 01$ in Fig. 2.2 instead, then the makespan of the mission consisting of three messages will be reduced.

In the execution of parallel algorithms, such as parallel state-space-search [92, 4], parallel sorting [104] and parallel Fourier-Transform [29], their communication behavior in a multicomputer network can often be characterized as a series of communication missions. In the case of missions containing heavy concurrent traffic, the network may become congested and turn into a bottleneck of execution efficiency. We will thus address the problem of message routing and scheduling in situations where there may be heavy concurrent communication traffic in the system, i.e., there exists a high possibility of contention for use of network resources. Our goal is to improve communication efficiency by implementing low-complexity, distributed message scheduling and routing in a distributed-memory system equipped with a communication adapter like SPIDER [31] at each node.

SPIDER (*Scalable Point-to-point Interface DrivER*) is a front-end hardware adapter that provides scalable communication support for point-to-point distributed systems. The microprogrammability of SPIDER enables a system to implement different topologies, routing algorithms, and switching methods. We will focus on the case where SPIDER is configured to work in k-ary n-cubes in general, and binary n-cubes in particular. Large-buffer message switching and virtual cut-through as well as circuit switching will be considered for the hypercube topology. Wormhole switching is found to be more suitable for high-radix low-dimension hypercube networks (such as meshes) with multiple virtual channels, and is covered in Chapter 3.

The work described here differs from others in several ways: (1) it stresses the importance of combining low-complexity adaptive routing with message scheduling as opposed to complex, fully-adaptive routing; (2) it zooms on the case of bursty traffic where a network is congested with concurrent communication traffic; and (3) it deals with the transient (rather than steady-state) performance of a network.

This chapter is organized as follows. Necessary notation and definitions are introduced

[1] Here the *arrival* of a message means it is ready to be sent out of the source node.

in Section 2.2. In Section 2.3, the proposed scheduling policies and routing algorithms are described. Section 2.4 deals with the performance evaluation of various scheduling–routing combinations. This chapter concludes with Section 2.5.

2.2 Notation and Definitions

The following notation will be used in this chapter.

- m_i: message i of length ℓ_i, $0 \le i < M$.
- M: the number of messages arrive in $[0, \Delta T]$.
- $[t, t + \Delta T]$: a *mission time frame* or time window during which messages of a mission arrive. Without loss of generality, we will assume $t = 0$ from now on.
- t_i^a: the arrival time of m_i, i.e., the time when m_i is ready to be sent.
- t_i^c: the completion time of m_i, i.e., the time when m_i reaches its destination.

For a set of M messages $\{m_i,\ 0 \le i < M\}$, each with arrival time t_i^a, and completion time t_i^c, the *makespan* of this set, $\hat{t}$, is defined as $\max\limits_{0 \le i < M}\{t_i^c\} - \min\limits_{0 \le i < M}\{t_i^a\}$. Given this set of messages to be exchanged among the nodes of a Q_n, we want to minimize this set's makespan and hence maximize link utilization of the network. Depending on the system implementation, we may achieve this goal with various combinations of message scheduling and routing.

A network with a mission arrival can be viewed as a physical system with a certain amount of injected *energy*, i.e., the total communication bandwidth required for the mission. $\hat{t}$ is essentially the time span required to *dissipate* the injected energy. Given the same mission, different message scheduling–routing combinations will affect the way the energy is dissipated in the system, thus resulting in different $\hat{t}$ values. For example, in Fig. 2.3, the remaining bandwidth of a mission is plotted against time for the hypercube under two different message scheduling policies. Although the initial bandwidth is the same, the *rates* of bandwidth decrease are different. Measuring the network performance using this approach is analogous to evaluating the transient response of an electronic component to step-function inputs. Similar to the case with an electronic component, as we shall demonstrate later, a network that performs well under this transient condition will perform well for most of the time under a steady-state condition.

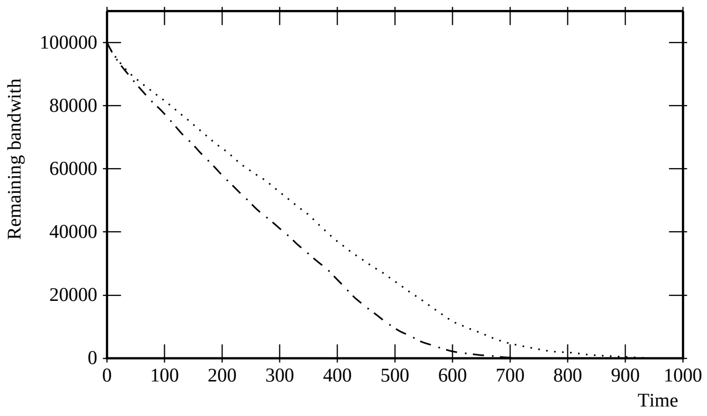


Figure 2.3: The remaining bandwidth against time under two different message scheduling policies.

2.3 The Proposed Message Routing and Scheduling Schemes

In this section we will describe several message scheduling policies and an adaptive routing algorithm that can be implemented with the aid of a SPIDER-like communication adapter. Note that in the presence of concurrent communication traffic, the system must use a traffic control mechanism that is efficient and doesn't cause excessive overhead. The schemes introduced here don't modify the underlying message format, and can all be implemented with minimal hardware, or, as in SPIDER, with a simple microprogram. Also, they don't need additional information on the task behavior except for the one already available in each message and each node.

2.3.1 Scheduling

We will evaluate the performance of several distributed message scheduling policies, while focusing on non-preemptive, low-complexity policies which utilize the existing message format. Each message has a *length field* as well as a *destination field*. The routing controller on a node can use the information in these two fields to determine which message to be sent first over each link of the node. The only major overhead comes from the implementation of a priority queue for each link, which can be implemented on a SPIDER-like adapter or by gate-level logic circuit. The time overhead can be ignored since in a SPIDER-like adapter, message queueing and communication can be done in parallel for non-preemptive scheduling.

In the default, FIFO, policy, no length or destination information is used, and the message at the front of a FIFO queue will be transmitted over a link. Also, in any of the following priority scheduling policies, a tie is broken by the FIFO principle. With the shortest-first (SF), and longest-first (LF) policies, messages in the queue of a link are arranged according to their lengths. When a message arrives and the outgoing link it requests is busy, it is entered into a priority queue. SF gives a shorter message higher priority, while LF awards higher priority to a longer message. The nearest-first (NF) and farthest-first (FF) policies take the destination field of a message and calculate the Hamming distance of the current node to the destination of the message. Entries in the priority queue are arranged in the order of increasing (as in NF) or decreasing (as in FF) Hamming distances.

The *remaining bandwidth* (RB) of a message on a node is defined as the product of the length of the message and the remaining Hamming distance to its destination. The *Smallest-RB-First* (SRBF) policy gives higher priority to smaller RB messages. The *Largest-RB-First* (LRBF) policy gives priority to larger RB messages. Implementing these two policies induces a slightly more overhead than the other policies due to a multiplication operation required before each message is entered into a queue. Again, with a SPIDER-like adapter, this overhead can be ignored since queueing and routing can be done in parallel.

Note that all of the above scheduling policies can be combined with the *Earliest Due-Date* (EDD) scheduling of message transmissions. With EDD scheduling, starvation can be avoided when a certain type of messages arrives continually. For example, with the LRBF policy, short messages may be blocked indefinitely if long messages arrive continually. This can be remedied by giving earlier arrivals higher priority.

The effectiveness of a scheduling policy can be measured by comparing its performance to that of optimal schedules. An optimal schedule is obtained by assuming that the following parameters are known *a priori*: t_i^a, ℓ_i, and the path each message is routed through. Thus, we have a classical scheduling problem in this case. For example, when message switching is used, by treating each link as a “processor”, and each message as a “job”, the optimization problem is essentially a special case of the job-shop scheduling problem [41], which is NP-hard. For other switching methods such as circuit switching and virtual cut-through, there are no known equivalent scheduling problems, but their computational complexity is also NP-hard. A branch-and-bound algorithm [13] is used to find optimal schedules. At each node in the search tree, we calculate an estimated bound to decide which node to be expanded next. The efficiency of the algorithm depends on the accuracy of the estimated bounds. In [108], we developed algorithms and equations to be used for computing bounds which are then used for finding optimal schedules for message switching, circuit switching, and virtual cut-through.

2.3.2 Routing

One major drawback of the e-cube routing algorithm is that the path chosen between a pair of processors is fixed regardless of the network traffic condition. When a certain link leading toward the destination of a message is busy, the e-cube algorithm simply holds the message and waits until the link becomes free, even if there can be an alternative free path. In situations where interprocessor communication traffic is light, this may not cause any serious performance degradation. However, under heavy traffic, it can become a major performance bottleneck. With distributed adaptive routing, a processor can select an alternative (free) link to route a message when the originally-selected link is busy. By doing this, it is possible that link utilization and network efficiency can be greatly enhanced.

There are numerous adaptive routing algorithms proposed for hypercubes [42]. Each scheme has its own advantages and disadvantages. More complex routing algorithms such as the DFS (Depth-First-Search) routing algorithm [18] require extra fields in each message to avoid livelock and achieve the backtracking capability. The major advantage of such algorithms is that all possible paths between source and destination nodes will be explored. However, their disadvantages include the overhead for storing information on detouring and backtracking, and more complex routing controllers. Furthermore, the information stored in a message can become out-of-date before it reaches the destination. For example, at a certain intermediate node, the routing algorithm may find all links that lead the message closer to its destination are busy, and hence decide to take a detour, i.e., a non-shortest path. But when the message is routed via a detour, one of the shortest paths may become available. This can result in a waste of network bandwidth, and hence degrade the performance. Also, unpredictable path lengths in this routing algorithm can result in inaccuracies in calculating the RB of a message, and therefore reduce the effectiveness of bandwidth-sensitive message scheduling policies such as SRBF and LRBF.

On the other hand, if we restrict the routing algorithm to the shortest paths only, albeit the full connectivity of the network is not explored, the above shortcomings can be avoided.

Besides, in a network equipped with SPIDER-like adapters, virtually no additional overhead is added to the default e-cube routing algorithm, since no modification needs to be done on the original message format, and the routing controller can be easily programmed with low-complexity code. Also, it is easy to have it work in tandem with any message scheduling policies since the length of each path is predictable. Characteristics of this distributed routing algorithm, called the *Progressive Adaptive* (PA) algorithm, are described as follows.

- As in e-cube routing, PA tries to route messages from the lowest dimension to the highest dimension based on the results of exclusive-ORing the source and destination addresses.

- When the link corresponding to the lowest dimension is busy, the next lowest dimension is tested, and so on, until a free link is found to route the message. If no link leading the message closer to its destination is available, the message is blocked and entered into a queue.

- Only one message queue is maintained on each node. All blocked messages are entered into the queue. Their order in the queue is determined by the underlying scheduling policy. When a link becomes free, the message closest to the head of the queue that can use the link to move closer to its destination is routed through this link.

The performance of the PA algorithm will be compared with DFS routing and a centralized path-selection (CPS) algorithm. As in the case of finding optimal schedules, the CPS algorithm operates on the premise that accurate message lengths are known *a priori*, and assigns paths to messages to balance the traffic in the network. With concurrent communication traffic, $\hat{t}$'s are usually dominated by the most heavily-loaded link. The problem of selecting a path in order to minimize $\hat{t}$ is equivalent to minimizing the maximum link load, and can thus be formulated as a mini-max optimization problem. This is a special case of Decision Problem 1 in [66], which was shown to be NP-hard. However, the true optimal solutions may not be meaningful since $\hat{t}$ is eventually determined by scheduling messages. Therefore, our goal is to find sufficiently good solutions that, when combined with the branch-and-bound scheduling algorithm, can achieve near-optimal performance. The CPS algorithm used is based on the simulated annealing method [72] and is described in [108].

2.4 Performance Evaluation

When the number of messages being sent concurrently into a network becomes larger, their interactions make the network behavior too complicated to predict and analyze. This calls for simulations to assess the performance of various scheduling–routing combinations. Our simulation model is summarized as follows:

- We assume processors — including their routing controllers — as well as communication links to be fault-free.
- Each communication link is half-duplex, i.e., at any instant of time, only one message can be sent in either direction of a link.
- As was supported in SPIDER, the routing controller on each processor can send or receive multiple messages at the same time, provided that the links needed are not in use. Also, incoming message buffering/queueing and outgoing message transmission are done in parallel for all switching methods.
- For circuit switching, the "call signal" for establishing a circuit is transmitted out-of-band, and doesn't interfere with the existing communication traffic.
- ΔT is relatively small, i.e., the communication traffic is highly concurrent. As a result, we assume there can be at most one message sent from node i to node j, $i \neq j$, within ΔT. If there are more than one message, they will be combined into a long message. We use Concurrent Communication Probability (CCP) to denote the probability that one processor sends a message to another processor in a mission. A higher value of CCP means heavier traffic. In the uniformly-distributed traffic pattern, ℓ_i is generated by the following routine:

  for $i := 0$ **to** $M - 1$
      for $j := 0$ **to** $M - 1$
          if $i \neq j$ **and** $rand1 <$ CCP **then** $\ell_i := rand2$;
          else $\ell_i := 0$;

  In the above pseudo code, $rand1$ is a uniformly-distributed random number in the interval $[0, 1]$, and $rand2$ is a normally-distributed random number with $\mu = 10$ and

$\sigma = 5$ and truncated to 0 when negative values are generated. In the *hot-spot* traffic pattern, the routine is similar except that the traffic is directed only to a given number of hot spots in a hypercube.

- In all of the presented data, we set $\Delta T = 0$. The results with $\Delta T > 0$ were found to be similar with the case of $\Delta T = 0$ combined with lower CCP values.

- Each data point is obtained by averaging the results of 10,000 iterations. Deviation from the mean values is found to be reasonablly small ($< 2\%$).

- The hypercube dimension used is 4. Results of this problem size are found to be typical among all tested sizes and are therefore selected for presentation.

2.4.1 Scheduling

Table 2.1 shows typical evaluation results of various scheduling polices. The data shown is obtained with CCP = 0.95 for uniformly-distributed traffic, i.e., a highly-congested condition. It is obvious that LRBF, with performance very close to the optimal schedules (OPT), outperforms the other policies.

	MS	CS	VCT
OPT	193.1	193.7	193.0
FIFO	231.1	229.0	218.5
LF	216.8	221.4	202.5
SF	240.5	233.7	221.4
FF	199.9	217.3	208.7
NF	244.4	232.7	219.5
LRBF	195.8	208.2	195.3
SRBF	250.1	231.5	226.6

Table 2.1: Performance of scheduling policies under the e-cube routing algorithm.

Our simulation results in evaluating the various scheduling policies are summarized as follows:

- SF, NF, SRBF are all worse than FIFO under all three switching methods, i.e., message switching (MS), circuit switching (CS), and virtual cut-through (VCT). Typically, SRBF has the worst performance among all the scheduling policies considered. It should be noted, however, that this is not the case with wormhole switching. As we will show in Chapter 3, SF, NF and SRBF outperform LF, FF and LRBF in a virtual-channel network with wormhole switching.

- FF has the performance characteristics closest to LRBF. In most situations FF outperforms LF, especially when the variance of message length is small. In message and circuit switching, FF generally outperforms LF, meaning that the distances to destinations have more pronounced effects. It is found that only under virtual cut-through, LF has performance closer to LRBF when the variance of message length is large. But as σ gets smaller, it gradually becomes closer to, and eventually gets outperformed by, FF.

- Under circuit switching, the performance differences are smaller for all distributed scheduling policies. The difference between the best and the worst are only ≈ 25 time units. Since messages never get buffered under circuit switching, distributed scheduling policies can only determine the order of sending messages at the source. It is expected that a distributed scheduling policy is less effective than a centralized policy in this case.

- Under virtual cut-through, messages are buffered only when a cut-through attempt fails, so the effects of distributed scheduling policies lie between those of message switching and circuit switching. However, given the same scheduling policy, virtual cut-through has consistently shown the best performance among the three switching methods. It is also interesting to note that as shown in Table 2.1, in case of heavy concurrent traffic, with a better scheduling policy such as LRBF, message switching can approach the performance of virtual cut-through.

- In all three switching methods, network performance generally gets better if message lengths are closer to uniform, i.e., σ is small.

In Figs. 2.4 to 2.6, the performance of LRBF scheduling is compared with the optimal schedules and FIFO scheduling under various CCP values for uniformly-distributed traffic. Figs. 2.7 to 2.9 show the comparison for a number of hot spots with CCP = 0.8. Under message switching, for CCP > 0.3, and in most cases of hot-spot traffic, LRBF improves significantly over FIFO and produces schedules whose makespans are within 10% of the op-

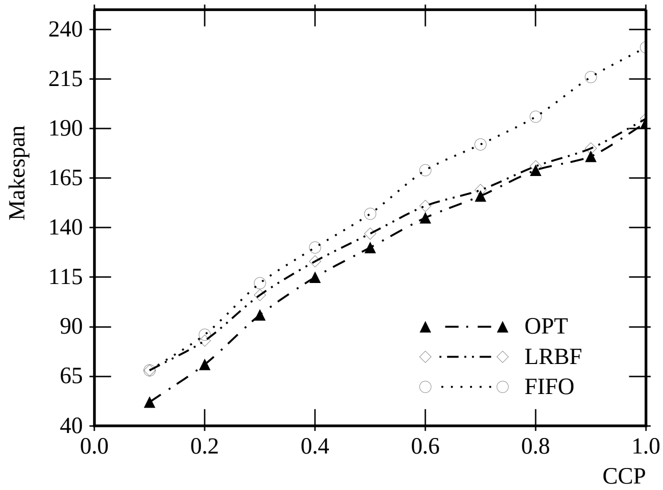


Figure 2.4: Uniformly distributed traffic, message switching.

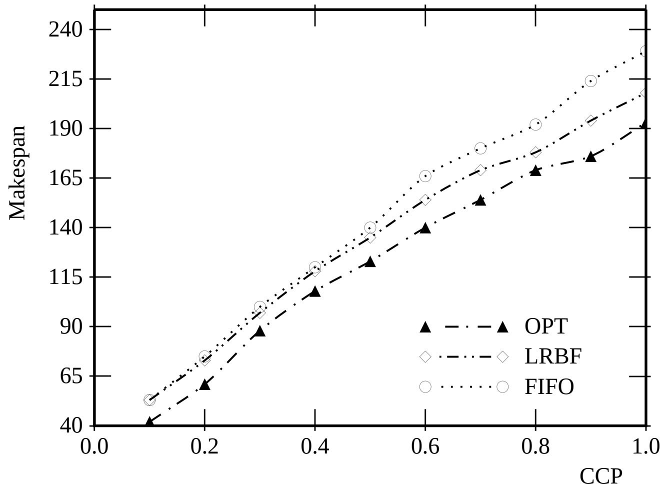


Figure 2.5: Uniformly distributed traffic, circuit switching.

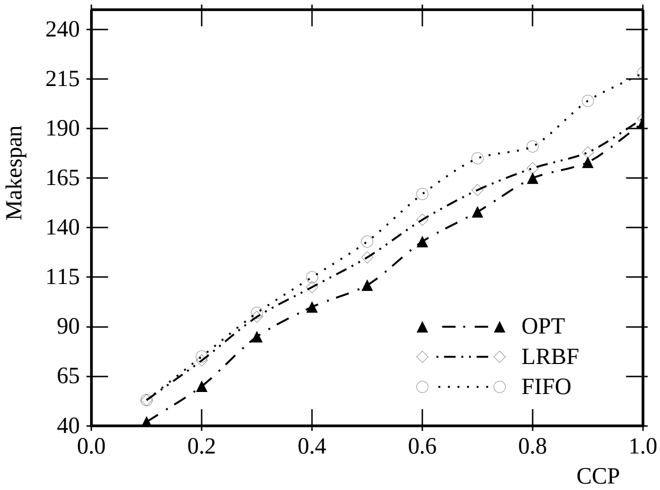


Figure 2.6: Uniformly distributed traffic, virtual cut-through.

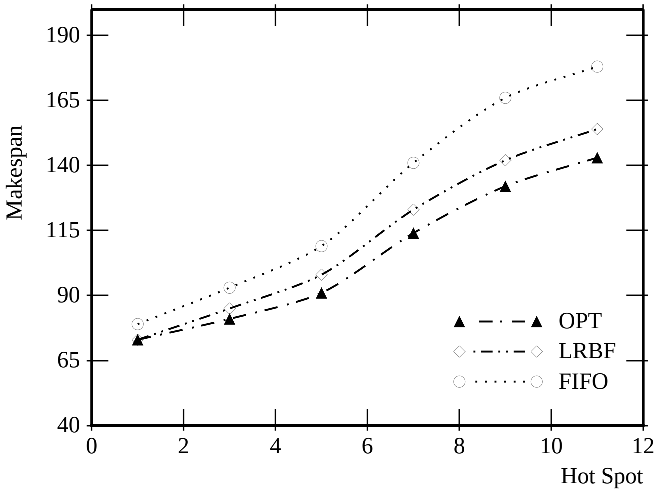


Figure 2.7: Hot–spot traffic, message switching.

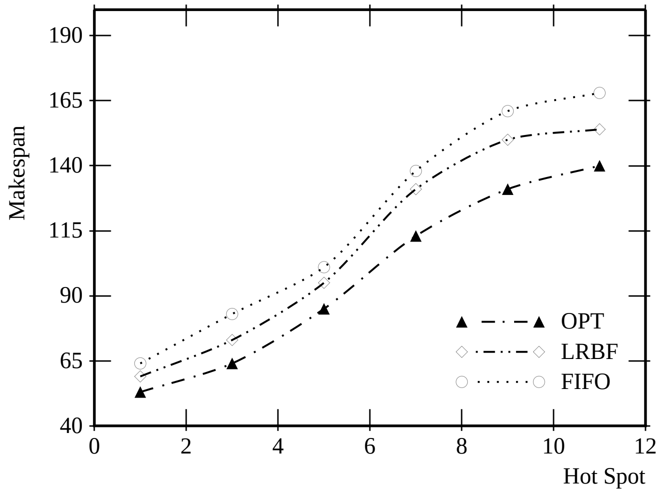


Figure 2.8: Hot–spot traffic, circuit switching.

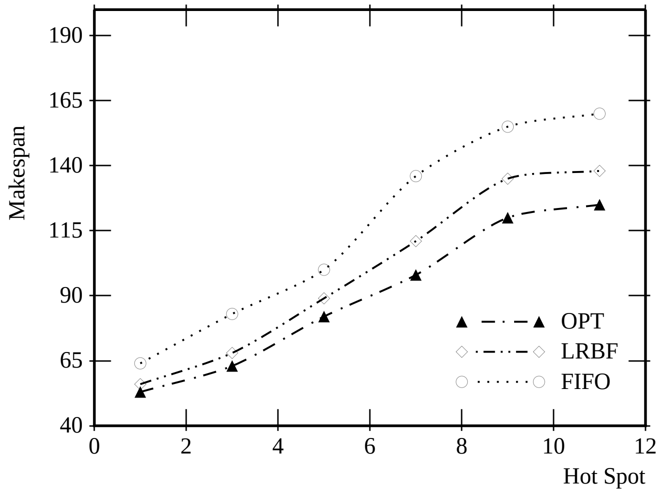


Figure 2.9: Hot–spot traffic, virtual cut-through.

timal schedules. Its performance is only slightly degraded under very light traffic, where all distributed scheduling policies become less effective and degenerate into FIFO scheduling. Performance of LRBF under virtual cut-through is very similar to the case under message switching, which is predictable since under heavy concurrent traffic, virtual cut-through essentially degenerates into message switching. In circuit switching, all distributed scheduling policies are less effective than in other switching since messages are never buffered and are scheduled only at the source nodes. Nevertheless, LRBF can still approach within 12% of optimal schedules for CCP > 0.5. The only case where LRBF does not improve significantly over FIFO is under circuit switching and hot-spot traffic.

2.4.2 Routing

The performance of PA routing combined with the LRBF scheduling is compared against the DFS routing working with the LRBF scheduling, the e-cube routing algorithm with FIFO message schedules (EQ-FIFO), and the CPS centralized path selection algorithm with optimal message schedules (CPS-OPT). The results are plotted in Figs. 2.10 to 2.12, for uniformly-distributed traffic, and Figs. 2.13 to 2.15 for hot-spot traffic with CCP = 0.8.

Our simulation results indicate that under all three switching methods, PA-LRBF significantly improves over the e-cube routing in all cases and outperforms DFS-LRBF in cases of heavy concurrent traffic. In most situations, it also approaches the performance of CPS-OPT closely within 12% in the message switching case, and in cases of circuit switching and virtual cut-through, within 7%.

It is only under light traffic condition (CCP < 0.4) or a very small number of hot spots (< 4) that the DFS routing has an advantage over the PA routing. It can approach CPS-OPT under very light uniformly-distributed traffic, and outperform CPS-OPT in case of a very small number of hot spots. However, under heavy traffic, the DFS algorithm does not fare much better than the e-cube routing, especially under message switching. This indicates that detouring and backtracking in the DFS algorithm have negative effects on network performance when the network is heavily congested. Besides, with the unpredictability of path lengths of the DFS routing, LRBF scheduling becomes less effective and nearly degenerates into FIFO scheduling.

The DFS algorithm performs significantly better under circuit switching or virtual cut-through than in the case of message switching. In fact, virtual cut-through with the DFS routing essentially degenerates into circuit-switching in our simulations. With both of

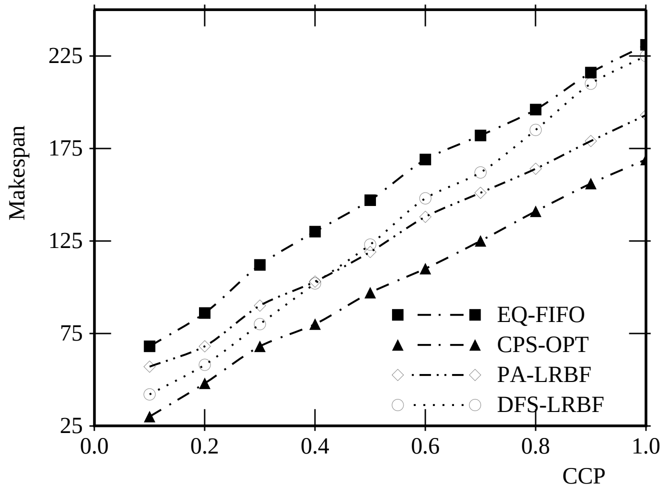


Figure 2.10: Uniformly distributed traffic, message switching.

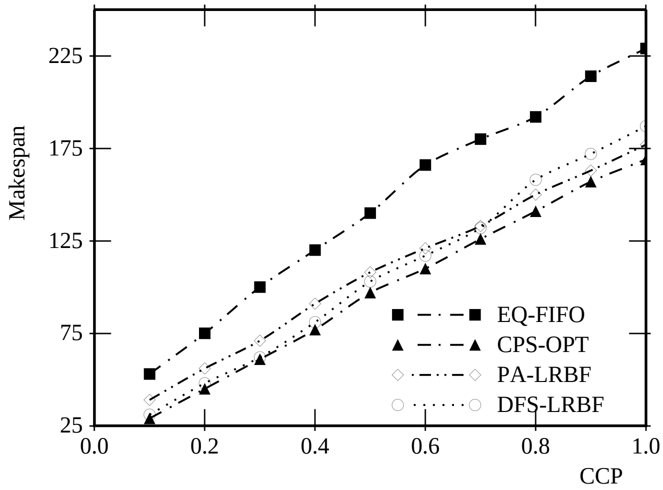


Figure 2.11: Uniformly distributed traffic, circuit switching.

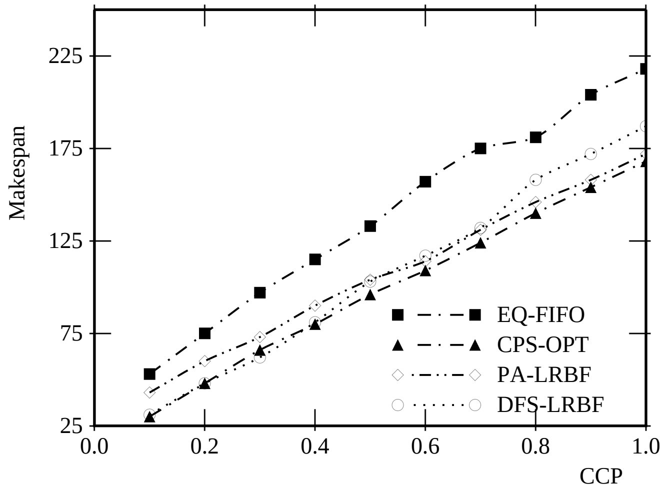


Figure 2.12: Uniformly distributed traffic, virtual cut-through.

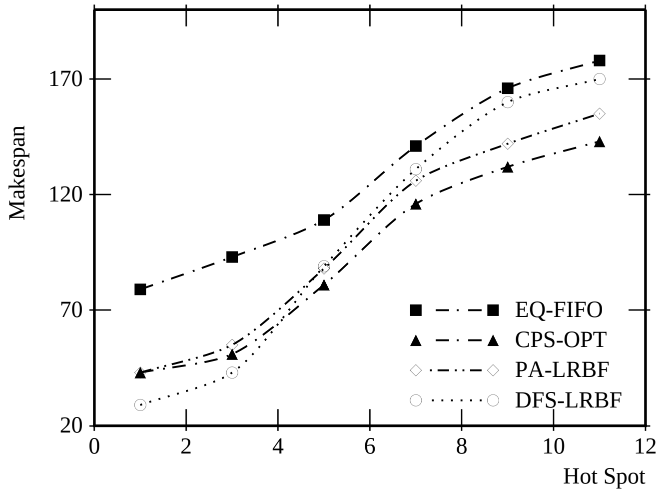


Figure 2.13: Hot-spot traffic, message switching.

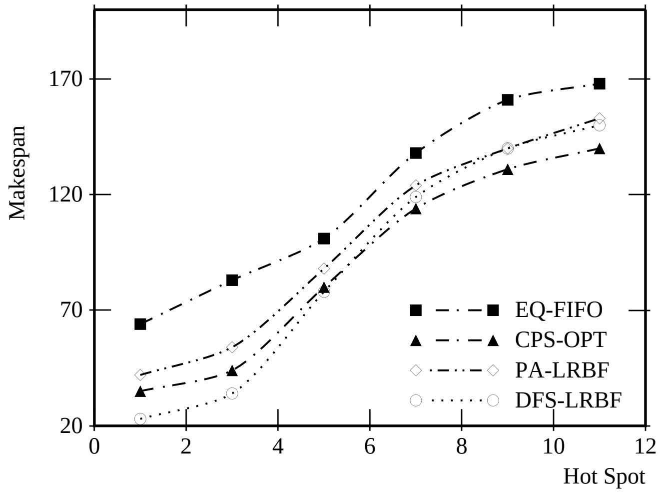


Figure 2.14: Hot-spot traffic, circuit switching.

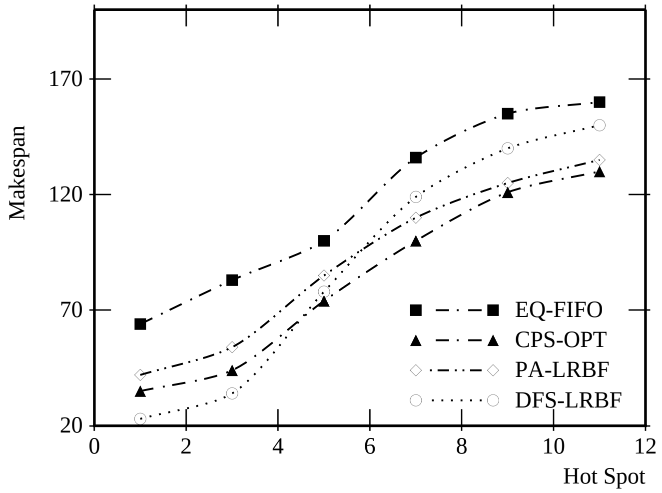


Figure 2.15: Hot-spot traffic, virtual cut-through.

these switching methods, the DFS algorithm either successfully routes a message to the destination, or blocks the message at the source. Therefore, there is no bandwidth waste in detouring and backtracking. While in the case of message switching where a message can be blocked at any node in the network, taking a detour and blocking a message at a node farther from its destination can degrade the performance. However, in implementing the DFS algorithm with circuit switching or virtual cut-through, the information on detouring and backtracking must be stored in the header used for establishing the path to avoid livelock. This can become a major overhead for large networks.

From the above results, one can conclude that, to improve network efficiency under heavy concurrent traffic, shortest-path adaptive routing combined with the LRBF message scheduling policy can approach the performance of near-optimal centralized routing and scheduling. Also, it is a more cost-effective alternative than a high-complexity fully-adaptive routing algorithm such as the DFS routing.

2.4.3 Steady-State Performance

Here we compare the performance of scheduling and routing mechanisms under steady-state traffic arrivals. Message arrivals at each node are assumed to follow a Poisson process. The mean latency of messages over a period of 20,000 units of time is plotted versus the mean message inter-arrival times. When scheduling messages, the *ages* of messages are also taken into account and combined with either FIFO or LRBF scheduling policies. That is, "older" messages are given higher priority to minimize the mean message latency and avoid starvation on continual message arrivals.

In Figs. 2.16 to 2.18, the performance of PA-LRBF, DFS-LRBF and EQ-FIFO are compared. Centralized schemes such as CPS-OPT are not included because their high computation cost makes them unsuitable for "on-line" applications in which message arrivals are continual. The steady-state results indicate that a scheme that performs well in transient conditions also performs well in a steady-state situation. Also, as in the transient case, the DFS algorithm outperforms the PA algorithm only under light traffic conditions.

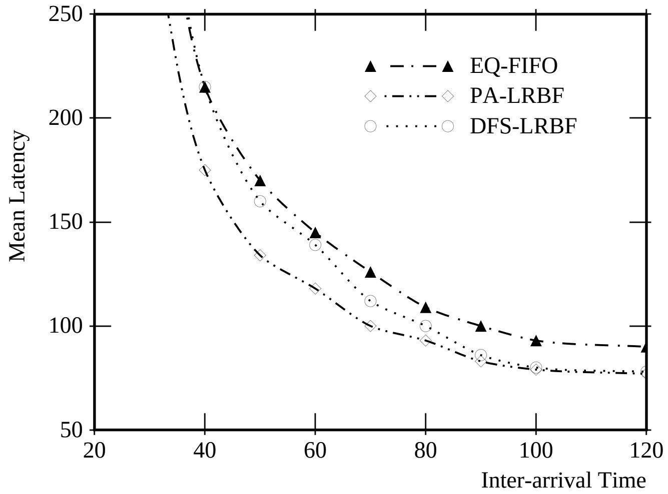


Figure 2.16: Steady–state performance under message switching.

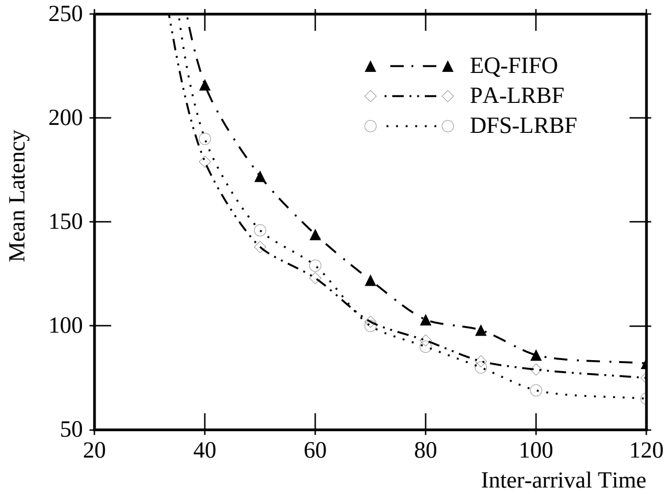


Figure 2.17: Steady–state performance under circuit switching.

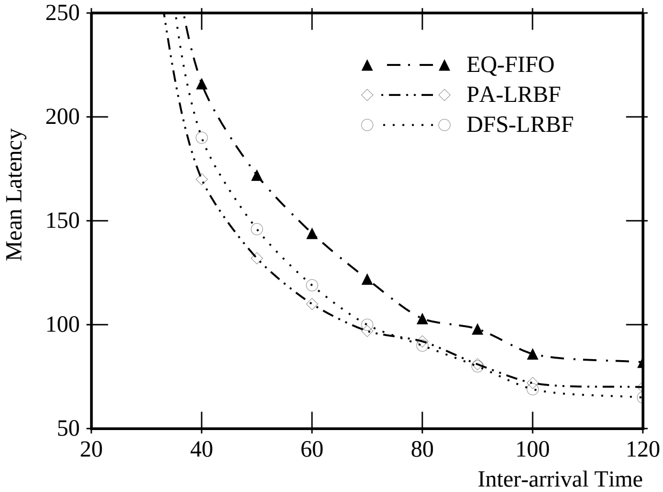


Figure 2.18: Steady–state performance under virtual cut-through.

CHAPTER 3

SEQUENCING OF CONCURRENT COMMUNICATION TRAFFIC IN A MESH MULTICOMPUTER WITH VIRTUAL CHANNELS

3.1 Introduction

The use of virtual channels multiplexed over each physical channel was introduced as a mechanism to accomplish deadlock-freedom by placing routing restrictions at intermediate nodes [24]. Virtual channels were also found to improve the network throughput via the increased sharing of each physical channel and the resulting reduction of message blocking [27]. That is, when there are multiple virtual channels per physical channel, messages of these virtual channels are allowed to time-multiplexed over the physical channel, thus blocking less number of messages (waiting for the physical channel to be available).

Pipelined-communication mechanisms, such as wormhole switching [24], operate based on the principle that the overall message latency can be reduced by pipelining the transmission of each message when the message must traverse multiple intermediate nodes. A message is broken up into small *flow-control digits* or *flits*, each of which serves as the basic unit of communication. The time taken for one flit to cross a physical channel is called the *flit time*. Header flits containing routing information establish a path through the network from the source to destination. Transmission of data flits is then pipelined through the path immediately following the header. A time-space diagram for wormhole switching is given in Fig. 3.1. Wormhole routing also has the advantage of requiring only a small on-line buffer space per node. While the pipelined nature of wormhole switching serves to reduce delivery latency, it may also propagate the effects of such bottlenecks as blocked flits and heavily-loaded physical channels. It is therefore important to devise a means of efficient allocation and management of network bandwidth.

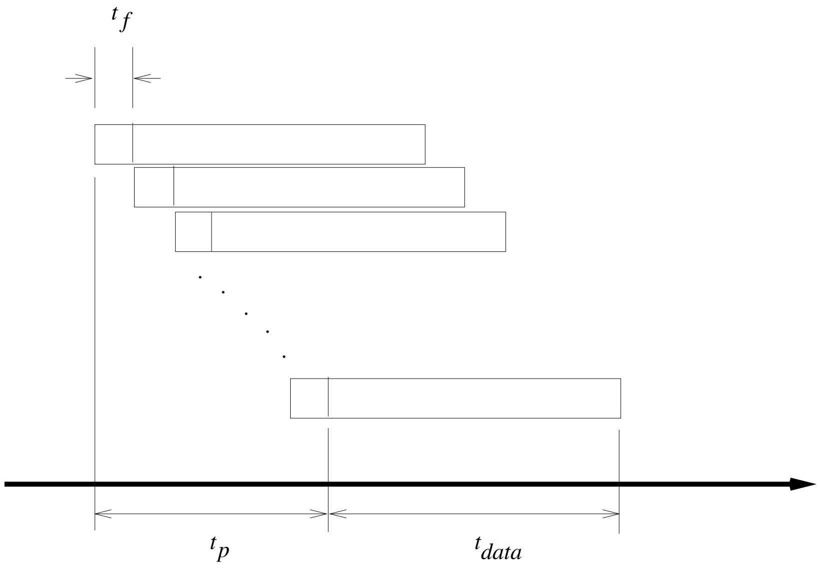


Figure 3.1: A time-space diagram of wormhole switching.

The network under consideration employs wormhole switching. Each pair of adjacent nodes are connected by a pair of uni-directional physical links/channels. A fixed number of uni-directional virtual channels are time-multiplexed into each physical channel. Though most of our discussion may apply to general networks, we will focus primarily on the mesh network topology, which has been widely used in evaluating the performance of virtual-channel networks [25, 26]. Especially, this chapter builds on the work by Dally [27] and Gaughan [42], where wormhole switching was found to significantly reduce message latency if it is combined with appropriate flow control schemes. We extend their work by focusing on bandwidth allocation through message scheduling and flit multiplexing.

In the previous related work [27, 42], communication traffic in a multicomputer network is often modeled as a number of mutually-independent, steady flows. However, this type of communication traffic does not always represent the real-world situation well, because network communication tends to be bursty. Message arrival[1] times are often clustered in a short period, which can temporarily saturate the network. Also, these messages may not be independent, and their delivery time as a whole is crucial to the overall performance. This tendency is exemplified by such algorithms as parallel sorting [104] and parallel Fourier-Transform [29].

In this chapter, we define a *communication mission*, or *mission* for short, to be a set of messages to be exchanged among the task modules which have already been assigned to

[1] Here the *arrival* of a message means that it is generated *and* ready to be sent out of the source node.

processing nodes in the network. During the execution of a parallel program, inter-node communication behaviors can be viewed as several independent communication missions. In addition to the usual mean latency, the *makespan* of a mission will also be used for performance evaluation. The makespan of a mission is defined as the maximum latency of all messages in the mission, i.e., the time span from the arrival of the first message until all the messages reach their destination.

As pointed out in [27], the most costly resource in an interconnection network is physical channel bandwidth, and the second most costly resource is buffer space. As shown in the following example, without proper message scheduling and flit multiplexing, increasing resources, such as physical and/or virtual channels, does not always improve performance, and may even degrade performance.

In Figs. 3.2(a) and (b), two messages A (light shaded) and B (dark shaded), each of length 4 flits, are sent over a physical channel to the same destination. We assume that one flit can be sent through the physical channel in a unit time. In Fig. 3.2(a), there is only one virtual channel per physical channel, and message A is given priority over message B. In Fig. 3.2(b), there are 2 virtual channels per physical channel, and hence, message transmissions are time-multiplexed over a physical channel. In this case, message B arrives at the same moment as in Fig. 3.2(a), but message A reaches its destination later by 3 unit time. So, in this example, adding one more virtual channel per physical channel actually *degrades* the performance.

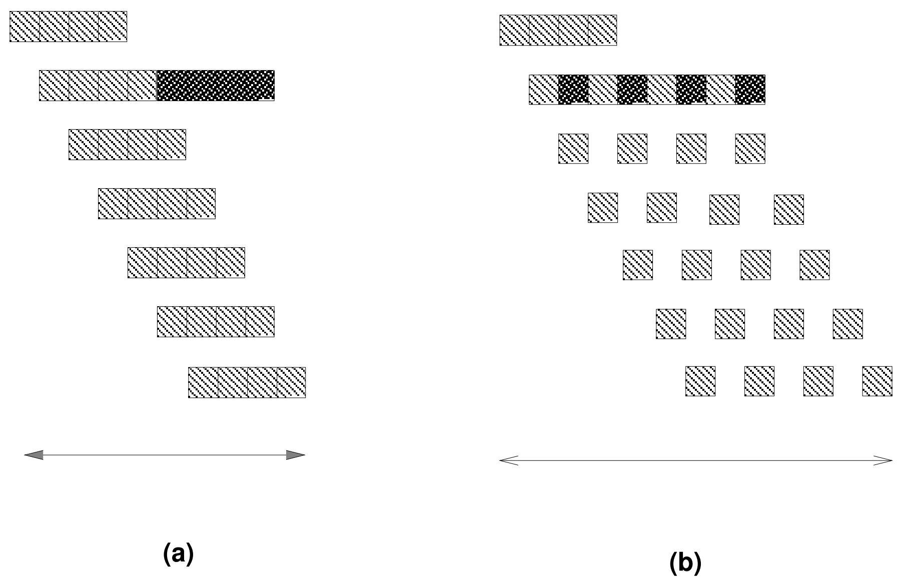


Figure 3.2: Wormhole routing with (a) single virtual channel (b) two virtual channels.

The main intent of this chapter is to (i) explore ways of sequencing messages and flits so as to better utilize network resources, and (ii) improve the overall network performance when more network resources are added. Especially, we will focus on the case when a substantial number of messages can be transmitted through the network *concurrently*.

The chapter is organized as follows. Basic terms and concepts necessary for our discussion are defined in Section 3.2. We formulate and analyze the problem in Section 3.3. Simulation results are presented and discussed in Section 3.4.

3.2 Preliminaries

A k-ary n-cube consists of k^n nodes arranged in an n-dimensional grid. Each node is connected to its Cartesian neighbors in the grid. For example, a 4-ary 2-cube is shown in Fig. 3.3.

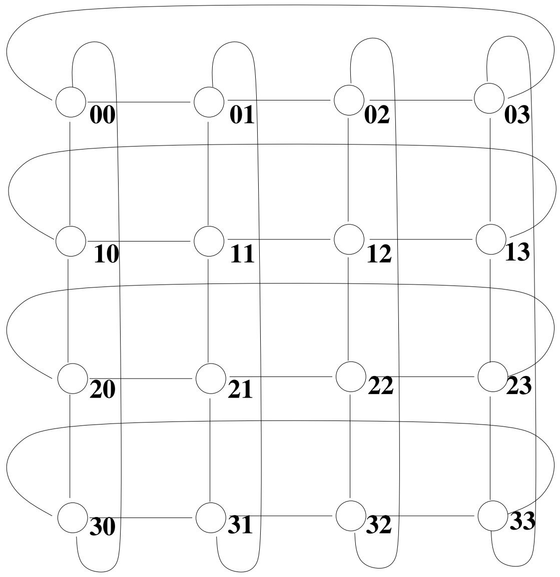


Figure 3.3: A 4-ary 2-cube.

A 2-dimensional $k \times k$ flat mesh is a subgraph of k-ary 2-cube, is not a regular graph, and has less edges than the corresponding k-ary 2-cubes (no wrap links at its boundary nodes). For convenience, we will call a k-ary 2-cube a *wrapped mesh*, or a *w-mesh* for short. Likewise, we will call a 2-dimensional flat mesh an *f-mesh*. Since an f-mesh is a subgraph of w-mesh with the same number of nodes, a w-mesh can also be made to function as an f-mesh by not using its wrap links.

Flow control in a virtual-channel network is performed at three levels: message routing, message scheduling, and flit multiplexing. Each of these can be implemented with a variety of algorithms, but we will consider only low-complexity, low-overhead flow-control mechanisms to deal with concurrent traffic in the network.

Routing: Selection of a path for each message. A message is routed to its destination via a fixed, shortest path. Issues related to fault-tolerance are not considered, or physical and virtual channels are assumed to be fault-free. In f-meshes, e-cube routing is used. The address of each node is expressed in terms of X and Y coordinates. A message is routed first in the X-direction until the Y coordinate of the node matches that of its destination node. It is then routed in the Y-direction. In w-meshes, a modified version of e-cube routing is implemented to utilize the extra communication links so that each message is routed via a shortest path. Deadlock-freedom is ensured by using the scheme proposed in [24]. That is, the virtual channels corresponding to each uni-directional physical channel are divided into high and low channels. Routing restrictions are then imposed such that either a high channel or a low channel, but not both, is allocated to each given message. The w-meshes need at least two virtual channels per physical channel to achieve deadlock-freedom.

Message Scheduling: Determining which message is allowed to access a free virtual channel in case of contention. When the number of messages to access a physical channel at the same time is larger than the number of available virtual channels, some of these messages have to be queued. So, we need to determine which messages are allowed to access the virtual channels, and which messages to be queued. We will consider the FIFO policy (as default), the *Largest Remaining Bandwidth First* (LRBF) policy, and the *Smallest Remaining Bandwidth First* (SRBF) policy. The *remaining bandwidth* of a message is defined as the product of message length and the distance from the current node to its destination. Note that, if all messages are of the same length, LRBF becomes farthest-first, and SRBF becomes nearest-first. Since we are dealing with concurrent traffic, an age-based policy such as deadline scheduling used in [27] is not meaningful and hence not considered. SRBF and LRBF can be easily implemented by using a priority queue instead of an FIFO queue. In case of non-uniform message lengths, a length field in the header flit of a message is added.

Flit Multiplexing: Determining the way messages are time-multiplexed over a physical channel. When there are multiple virtual channels per physical channel, the messages allocated to these virtual channels are multiplexed over the physical channel. Flit multiplexing determines the order for these flits from different virtual channels to access the physical channel.

In the default, *Round-Robin*(RR), multiplexing, virtual channels take turns in accessing the physical channel without using any network or message information. RR multiplexing without any modification will henceforth be called *strict* RR. Like message scheduling, flit multiplexing can be priority-based. The simplest priority-based multiplexing is called the *greedy* policy. Under the greedy policy, if a virtual channel had successfully sent a flit in the previous cycle, it will be given the highest priority again to access the physical channel in the next cycle. Otherwise, the access right is rotated to the next virtual channel.

More complex multiplexing can also be built based on a message's remaining bandwidth requirement. The *Largest Remaining Bandwidth Preferred* (LRBP) multiplexing method awards priority to the virtual channel containing a message of larger remaining bandwidth requirement. By contrast, the *Smallest Remaining Bandwidth Preferred* (SRBP) multiplexing method gives priority to the one of smaller remaining bandwidth requirement. As pointed out in [27], these multiplexing methods can all be implemented with combinational logic which operates on the contents of the status register associated with each virtual channel. The added hardware cost should not be a concern if the number of virtual channels is not too excessive.

If each virtual channel is allocated a fixed physical bandwidth regardless of whether the virtual channel is in use or not, this can lead to a substantial waste of physical bandwidth. *Demand-driven*(DD) allocation can be used to rectify this problem. With DD allocation, virtual channels will contend for use of a physical channel only if they have flits to send. DD allocation can be easily implemented by adding low-complexity combinational circuit to any multiplexing method.

With *CTS (Clear-To-Send) lookahead*, virtual channels only contend for use of a physical channel if each of them has a flit to send *and* the receiving node has room for it. This can further reduce the waste of physical bandwidth. When CTS lookahead is implemented, the receiving-end of each virtual channel must send a status bit back to the sending-end. These signals can be sent via separate wires [24], which requires extra hardware. Or they can be sent over the physical channel in the opposite direction, which can result in a non-negligible bandwidth overhead.

3.3 Formulation and Analysis

In this section, we discuss the tradeoffs among different message-scheduling policies and flit-multiplexing methods under the following assumptions.

- A physical channel takes one unit of time to transmit a single flit. A unit of time will also be called a physical-channel *cycle*.
- There is a single-flit buffer associated with each virtual channel.
- A message arriving at its destination is consumed without waiting.
- There are an even number of virtual channels associated with each physical channel in a w-mesh.
- For simplicity, the *time window* of message arrivals, ΔT, is assumed to be 0.

The following notation will be used in our discussion.

- d_{ij}: the length of the shortest path from node i to j.
- h_{ij}: the Hamming distance from node i to j, obtained by summing the difference of addresses in each dimension. Note that in w-meshes $h_{ij} \geq d_{ij}$, while in f-meshes they are equal.
- k_{ij}: the maximum number of messages that share a physical channel in the path from node i to j.
- ℓ_{ij}: the length of m_{ij} in number of flits.
- m_{ij}: the message to be sent from node i to j.
- v: the number of virtual channels per physical channel.

The *latency* of m_{ij}, denoted as t_{ij}, is the time span from a message's arrival to acceptance of the last flit of the message by its destination. We will use $\bar{t}$ to represent the mean latency of a mission. Given $\Delta T = 0$, the *makespan*, denoted as $\hat{t}$, of a communication mission is the maximum latency of all messages in the mission. We will evaluate the performance of a network with both the mean mission latency and makespan. Formally, we have

$$t_{ij} = t_{ij}^0 + (1/r_{ij})(\ell_{ij} - 1).$$

The first term, t_{ij}^0, denotes the time span between the arrival of m_{ij} at the source node i and the arrival of its header flit at the destination node j. t_{ij}^0 is composed of two components: accumulated queueing delay t_{ij}^q and accumulated head flit-multiplexing delay t_{ij}^x. t_{ij}^q is the sum of queueing times at all nodes in the path waiting for an available virtual channel. t_{ij}^x is the sum of times m_{ij}'s header flit waits at all nodes on its path for use of physical channels. The second term, $(1/r_{ij})(\ell_{ij} - 1)$, represents the time required for all other flits of m_{ij} to arrive at node j, which is determined by ℓ_{ij} and the transmission rate, r_{ij}, of the pipeline set up for m_{ij}. Depending on the flit-multiplexing method used, r_{ij} may change with time during a mission.

Message Scheduling

Given a communication mission and fixed v, t^q will be affected by the underlying message-scheduling policy. Under the LRBF policy, messages requiring larger bandwidths are given priority. Since those messages farther away from their destinations are more likely to have larger t^x's, by minimizing their t^q's, we may minimize the variance of message latencies. Similarly, the second term of t_{ij} is larger for longer messages. By giving these messages higher priority in using virtual channels, the balancing effect of smaller t^q's and hence smaller t^0's can also minimize the variance of message latencies. However, in wormhole switching, a blocked message does not release resources already allocated to it. A message farther away from its destination is more likely to be blocked and may therefore result in more resources being held. Also, a longer message can hold up resources in the path for a longer time, thus blocking more of the other messages. Under the SRBF policy, messages requiring smaller bandwidths are given higher priority. This heuristic is proposed based on the conjecture that if these messages are delivered to their destinations quickly then there will be less number of messages contending for communication resources and the overall concurrent communication traffic can be reduced. Thus, the remaining messages may be sent through the network encountering less contention.

One can easily observe that the effect of message-scheduling policies will be more pronounced with a small number of virtual channels. If v is large, then most of the contending messages will be allocated virtual channels, thus making the effect of message-scheduling policies less pronounced.

Flit Multiplexing

Generally, t^q decreases with the increase of v. However, under strict RR multiplexing without DD allocation or CTS lookahead, $r_{ij} = 1/v \; \forall i, j$. So, $1/r_{ij}$ increases with v and t^x may increase because a flit may need to wait more cycles for use of the physical channel. That is, there is a tradeoff between t^q and r_{ij}, and also between t^q and t^x, when more virtual channels are added. Usually when v is increased to a certain point, the improvement in reducing t^q reaches a plateau, and adding more virtual channels only increases message latency.

In case of strict RR, a large portion of physical bandwidth can be wasted on idle virtual channels. With DD allocation (denoted as DD-RR), r_{ij} is bounded below by $\max\{1/v, 1/k_{ij}\}$. In the worst case, $r_{ij} = 1/v$ and latency t_{ij} is the same as in the case of strict RR. But, if at any instant there are less than v messages contending for a physical channel, then no physical channel will be allocated to any idle virtual channel. Messages can be transmitted at a rate $\geq 1/v$. When other multiplexing methods are used, r_{ij} is not a constant, and hence, it is not easy to predict the performance.

Even with DD allocation, physical bandwidth can still be wasted if the corresponding input buffer is not ready to receive a new flit. With CTS lookahead, a virtual channel will not contend for a physical channel unless it has a flit ready to be sent and the receiving end has room for accepting it. For a given multiplexing method, (r_{ij} |with neither DD nor CTS) $\leq$ (r_{ij} |with DD) $\leq$ (r_{ij} | with DD and CTS).

Another advantage of CTS is deadlock-freedom. In priority-based flit multiplexing, a deadlock may occur if the method is not carefully implemented. For example, in Fig. 3.4, each physical channel has two virtual channels. All messages are routed to the same direction on node 2. Messages A and B have established pipelines from node 0 to node 2, and node 1 to node 2, respectively. So on node 2, the two messages have occupied both of the output buffers in the direction which all messages need to be routed to. Suppose message C arrives at node 0 later than A, and occupies the other virtual channel, and the same situation occurs on node 1 when message D arrives after B. If C is given priority over A, then C will access the physical channel between node 0 and 2. Similarly, D also has a higher priority than B, and monopolizes the physical channel between node 1 and 2. But on node 2, both of the output buffers are occupied by A and B, and they cannot be preempted since wormhole switching is used. Therefore, C and D will be queued at node 2 indefinitely waiting for free output buffers, while A and B cannot access the physical channels which they need to finish sending their remaining flits and release the output buffers on node 2 that C and D are waiting for. So, a deadlock follows.

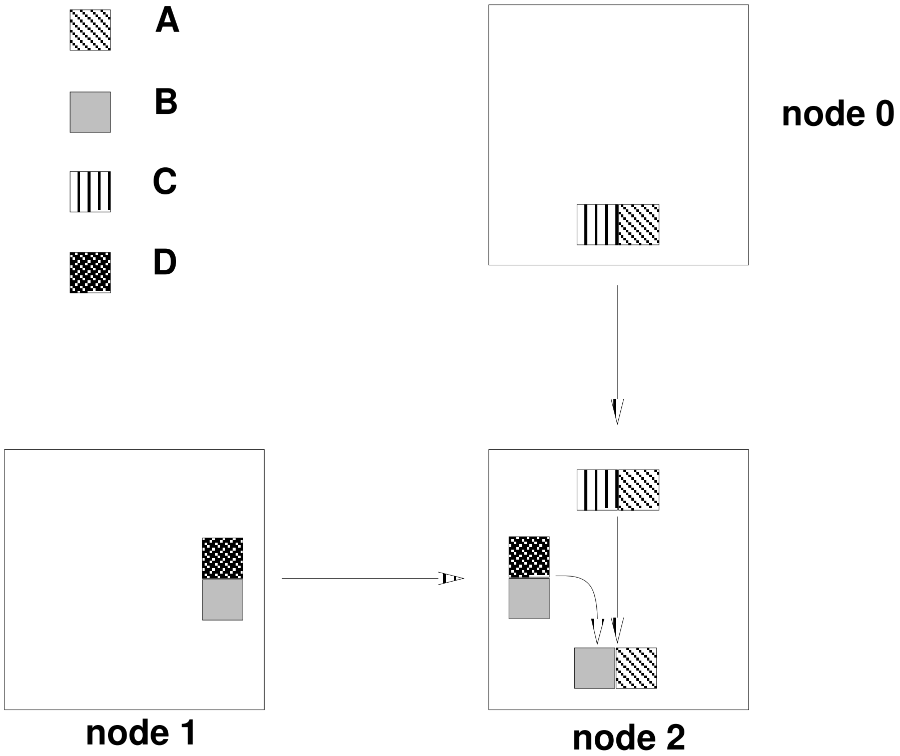


Figure 3.4: An example deadlock caused by priority-based flit multiplexing.

If CTS lookahead is used in the above example, then one can avoid the deadlock. Since C and D will not be allowed to contend for physical channels, A and B will be able to continue sending their remaining flits. In this chapter, we will always evaluate LRBP and SRBP flit multiplexing with CTS lookahead to avoid deadlock.

W- and F- meshes

When compared to an f-mesh with the same number of nodes, a w-mesh has the advantages of more physical channels and smaller communication diameter. Also, its regular connectivity may lead to better communication load balancing, especially in the case of uniformly-distributed traffic.

However, one main drawback of w-meshes is the potential deadlock resulting from the addition of wrap links. To ensure deadlock-freedom, virtual channels running over each physical channel must be divided into two halves. When messages are sent between a pair of nodes between which the Hamming distance is $< k/2$ in a k-ary 2-cube, only $v/2$ virtual channels are available. If the other $v/2$ virtual channels are not in use, they are left idle and their bandwidth wasted. Thus, there exists a tradeoff between these two topologies. Depending on traffic density and distribution, one can outperform the other. Our simulation results in the next section demonstrate this tradeoff.

3.4 Simulation Results

Under the following assumptions, we developed a program that simulates the flit-level communication behavior of virtual-channel networks.

- Transferring a flit between two nodes via a physical channel takes one unit of time.
- At any instant of time, all flits that have been allocated channels are transferred synchronously in a single physical channel cycle.
- Each virtual channel is assigned a single-flit buffer.

In the discussion that follows, the term *configuration* is used to represent a combination of some message-scheduling policy with a flit-multiplexing method. The simulation results presented here were obtained using the following parameters:

- Both w- and f- meshes are of size 16×16.
- Unless stated otherwise, all messages are 20 flits long. Traffic is uniformly-distributed. For a given mission, the probability that node i may send a message to node j is fixed. The Concurrent Communication Probability (CCP) that node i sends a message to node j is 0.01. In a 16×16 network, the total number of concurrent messages during a mission is $\approx 0.01 \cdot (16^2 - 1)^2$. Results on inputs with variable message lengths and other traffic patterns are not presented. General trends of the results from these alternative inputs do not deviate significantly from the data shown.
- Each data point is obtained by averaging results from 10,000 iterations. Deviation from the mean values is found to be $< 5\%$.

Table 3.1 shows the makespan ($\hat{t}$) and the mean latency ($\bar{t}$) of w- and f- meshes with the FIFO message-scheduling policy and strict RR flit multiplexing. Clearly, with strict RR, w-meshes not only perform worse than f-meshes with the same v, but also worse than f-meshes with $v/2$ virtual channels. Thus, addition of physical and virtual channels in w-meshes actually degrades the performance if strict RR is used.

	$\hat{t}$		$\bar{t}$	
v	w-mesh	f-mesh	w-mesh	f-mesh
1	n/a	801	n/a	280
2	1389	652	470	228
4	1090	546	345	209
6	945	508	305	223
8	812	502	305	251
12	707	533	341	304
16	688	629	377	414

Table 3.1: Performance of w-meshes and f-meshes with strict RR flit multiplexing

Table 3.2 shows the case of DD allocation. Obviously, DD allocation greatly improves the performance, particularly in the case of w-meshes with larger v's. In a certain situation, $\hat{t}$ and $\bar{t}$ are reduced by more than 50%. With f-meshes, DD also makes a monotonic improvement of $\hat{t}$ with the increase of v. Note that with DD-RR, w-meshes start to have smaller makespans than f-meshes with the same number of virtual channels when $v \geq 8$. Nevertheless, in the case where both types of network have equal number of nodes, we have to take into account that w-meshes have more physical channels. In a network of 16×16 nodes, a w-mesh has 1024 uni-directional physical channels while an f-mesh has only 960. Considering this, w-meshes still have poorer physical channel utilization, even with $v \geq 12$.

From the results in Tables 3.1 and 3.2, we conclude that physical bandwidth is used much more efficiently with DD-RR than strict RR. Since DD allocation can be implemented with minimum hardware overhead over any configuration, all configurations will be evaluated with DD allocation. DD-RR with FIFO will be used as our "default" configuration. The data in Table 3.2 will be used as the reference for other configurations. The makespan and the mean latency of each configuration are plotted versus v.

Fig. 3.5 shows the comparison of the LRBF and SRBF message-scheduling policies with FIFO in w- and f- meshes. DD-RR multiplexing is assumed in all three configurations. In w-meshes, SRBF shows a significant improvement ($> 30\%$) over FIFO for $v = 2$. The margin of improvement decreases with larger v's, which drops below 10% when $v > 8$, and reduces to near 0% after $v > 12$. The sharp drop after $v > 6$ can be attributed to the fact that, when $v > 6$, adding virtual channels improves FIFO, and makes the effect of

	$\hat{t}$		$\bar{t}$	
v	w-mesh	f-mesh	w-mesh	f-mesh
1	n/a	801	n/a	280
2	889	616	316	216
4	635	507	200	188
6	516	457	162	189
8	401	432	147	198
12	310	418	144	216
16	290	410	151	229

Table 3.2: Performance of w-meshes and f-meshes with DD-RR.

message scheduling less significant. On the other hand, LRBF does not make any notable improvement over FIFO. Only when $v = 2$ it shows $\approx 8\%$ improvement.

In f-meshes, SRBF shows 10% to 17% improvements over FIFO for $v = 1$ and $v = 2$, and quickly drops to the same with FIFO when $v \geq 6$, while LRBF is only marginally effective when $v = 1$ ($\approx 7\%$ improvement) and is virtually the same with FIFO for any $v \geq 4$.

Fig. 3.6 compares the mean latency, $\bar{t}$, of the three message-scheduling policies. In w-meshes, SRBF reduced the mean latency by more than 30% for $v = 2$ but the margin reduces gradually down to less than 10% when $v = 6$ and later drops to near 0% after $v \geq 8$. Similarly, in f-meshes, SRBF is effective for a small number of virtual channels($v = 1$ and $v = 2$), reducing $\bar{t}$ by at least 12%. But it performs virtually the same as FIFO for $v \geq 4$. In both topologies, LRBF makes virtually no improvement over FIFO in terms of $\bar{t}$ at any value of v and is not shown in the plot.

Figs. 3.7 and 3.8 show how the three message-scheduling policies perform when combined with the greedy flit-multiplexing method. The data of LRBF-G is omitted since its performance is very close to FIFO-G.

In w-meshes, greedy multiplexing clearly reduces $\hat{t}$ in all three configurations for $2 \leq v < 6$ by almost 30% when $v = 2$. However, when $v > 6$, greedy multiplexing degrades their performance. This is due to the nature of greedy multiplexing that awards access of physical channels to the same virtual channel that succeeded in sending its message last time. This can be beneficial in reducing the makespan of a mission when v is small, but works against the case with a large number of virtual channels since the time-multiplexing

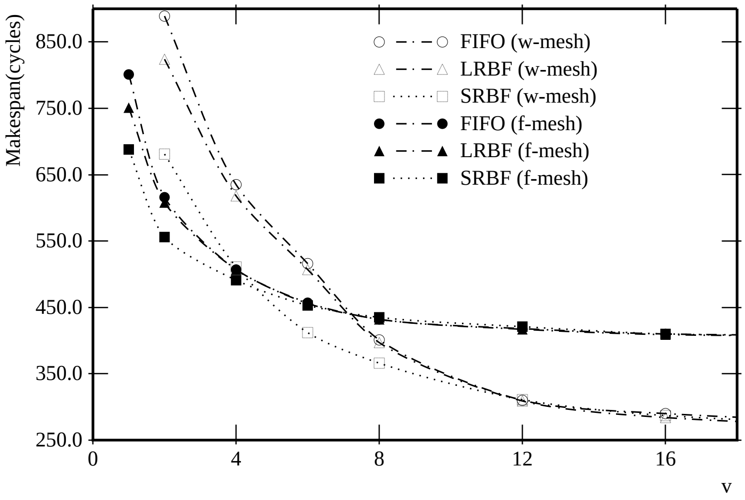


Figure 3.5: Makespan comparison.

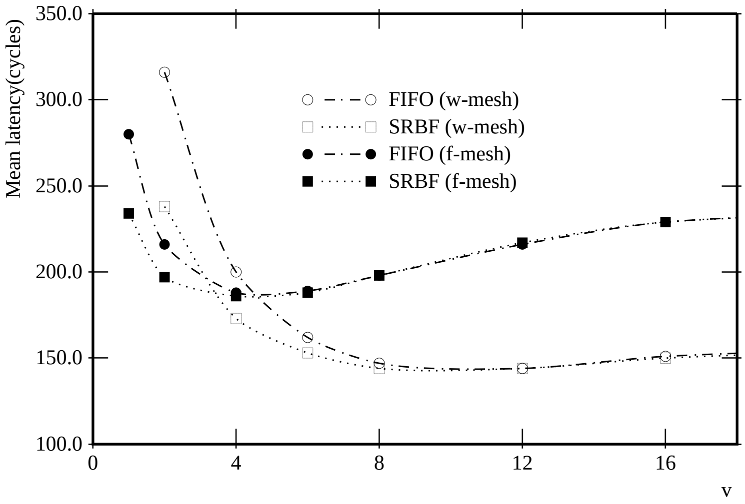


Figure 3.6: Mean latency comparison.

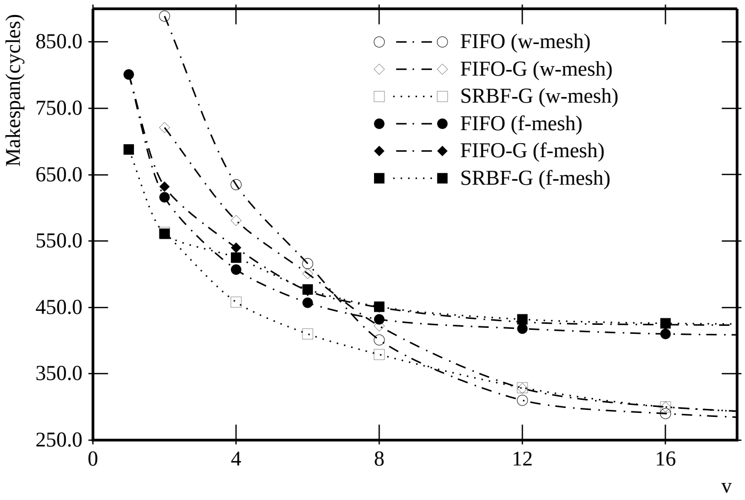


Figure 3.7: Makespan comparison.

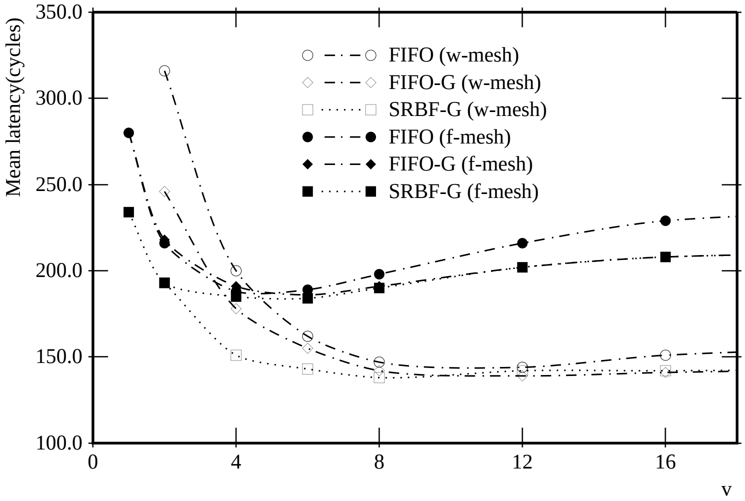


Figure 3.8: Mean latency comparison.

effect is reduced. In f-meshes, greedy multiplexing is not effective in improving performance over RR multiplexing. It actually results in larger $\hat{t}$'s for all three configurations in most cases.

In terms of $\bar{t}$, greedy multiplexing improves SRBF significantly for $v \leq 4$ and $v > 12$ in w-meshes. In f-meshes, it is also quite effective in reducing $\bar{t}$'s when $v > 8$ with all three message-scheduling policies, showing nearly 12% reduction of $\bar{t}$. Thus, as characterized by a typical increase of $\hat{t}$ and decrease of $\bar{t}$ when v is large, greedy multiplexing tends to increase the variance of message latencies. Also, note that SRBF-G, LRBF-G and FIFO-G perform almost the same with larger v's, demonstrating that the effect of flit multiplexing takes over message scheduling.

CTS lookahead can effectively minimize the waste of physical channel cycles, but its higher implementation cost may not be justifiable if the margin of improvement is small. In [42], it is shown that with pipelined circuit-switching, CTS is not very effective. As for wormhole switching, it was shown in [27] that in a network with 32-bit flits and $v = 15$, without adding extra wires for the lookahead signals, an additional 12.5% traffic overhead is required to implement CTS lookahead. Therefore, CTS lookahead should provide at least 12.5% improvement to justify its implementation overhead.

The effects of adding CTS lookahead on the three message-scheduling policies are plotted in Figs. 3.9 and 3.10. CTS lookahead is very effective in w-meshes, reducing $\hat{t}$ over the corresponding non-CTS lookahead version for 10% to 30% when $2 \leq v \leq 12$, and $\bar{t}$ for at least 15% when $2 \leq v \leq 8$. CTS lookahead is less effective in f-meshes, however. The greatest improvement ($\approx$ 10%) in $\hat{t}$ over non-CTS versions occurred when $v = 4$. The improvement margin gradually decreases down to 0% when v is increased to 16. It reduces $\bar{t}$ by at most 10% when $v = 2$ and $v = 4$. The effect of combining greedy multiplexing with CTS lookahead is not significant in terms of $\hat{t}$. Nevertheless, greedy multiplexing is still effective in reducing $\bar{t}$ for large v's, in both w-meshes and f-meshes.

The effects of SRBP and LRBP multiplexing (with CTS lookahead to avoid deadlocks) are plotted in Figs. 3.11 to 3.14. In w-meshes, as compared to the corresponding configurations with CTS lookahead only, the SRBP version further reduces $\hat{t}$ by $\approx$ 10% when $4 \leq v \leq 8$. But the margin of improvement gradually decreases when v reaches 12, and actually has worse performance for larger v's. In f-meshes, SRBP multiplexing improves only slightly the CTS-only version. The maximum margin of improvement occurs at $v = 4$.

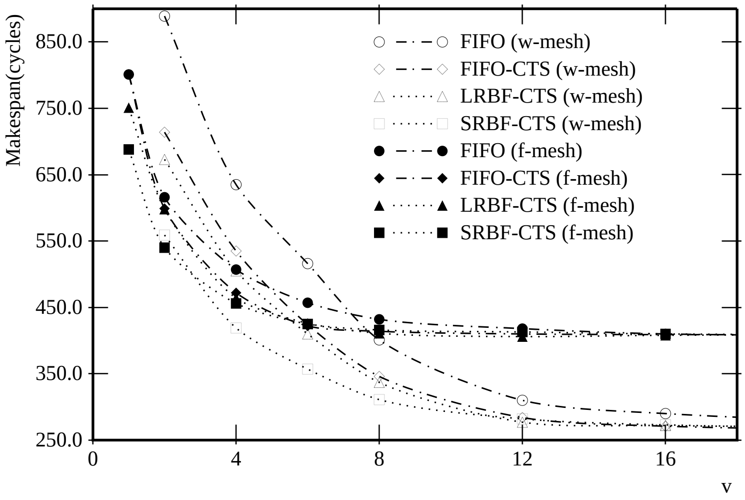


Figure 3.9: Makespan comparison.

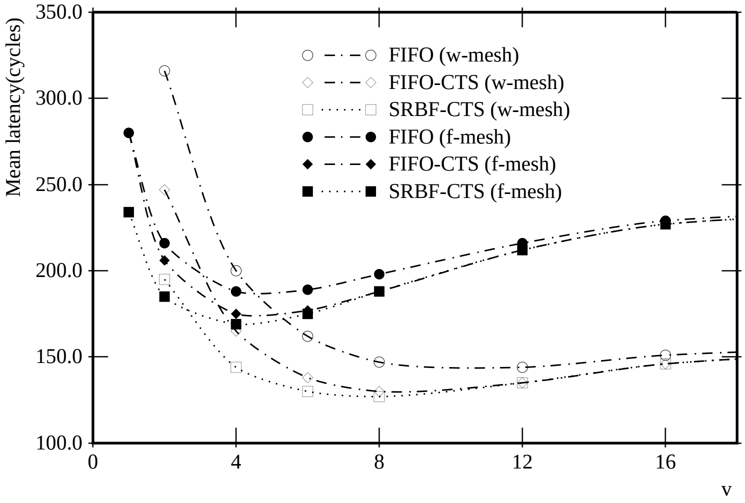


Figure 3.10: Mean latency comparison.

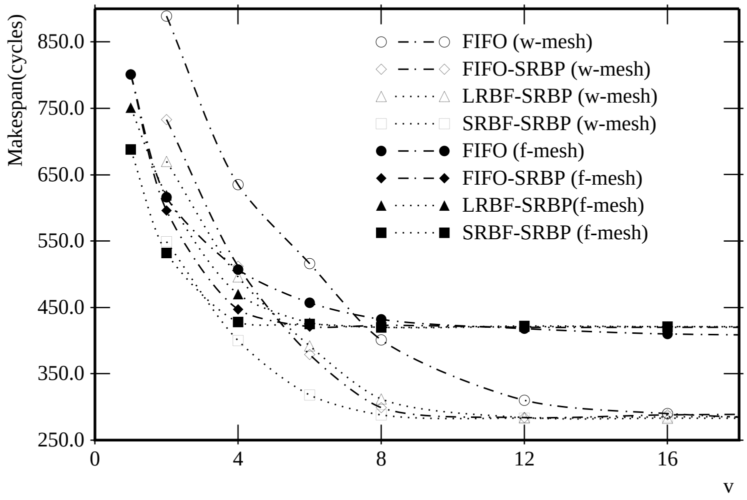


Figure 3.11: Makespan comparison.

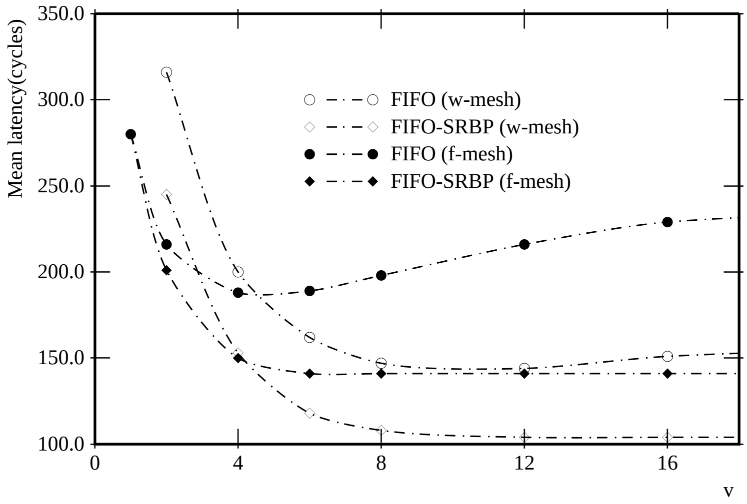


Figure 3.12: Mean latency comparison.

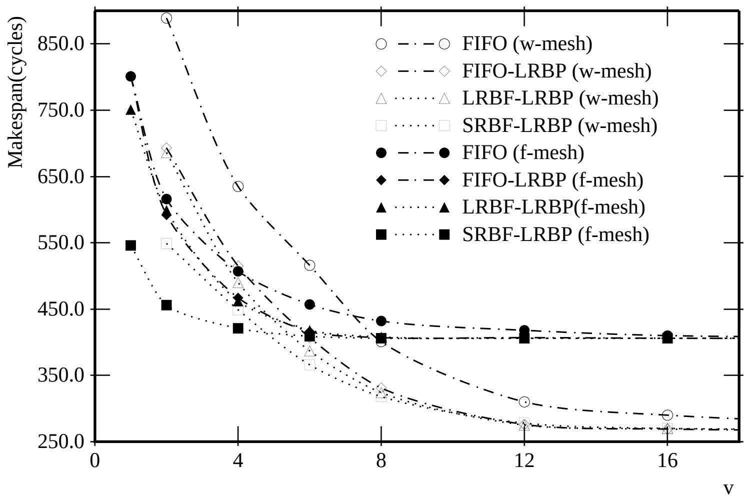


Figure 3.13: Makespan comparison.

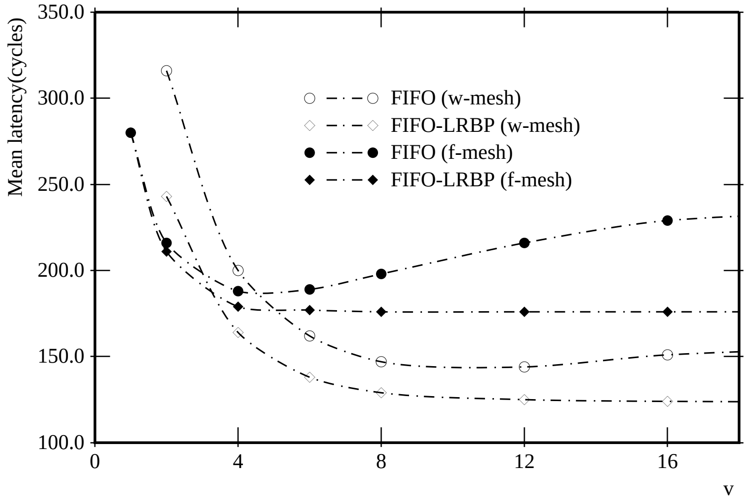


Figure 3.14: Mean latency comparison.

SRBP multiplexing is extremely effective in reducing $\bar{t}$. As can be seen in Fig. 3.12, for $2 \leq v \leq 32$, this multiplexing method reduces $\bar{t}$ significantly for FIFO, and clearly outperforms the CTS-only counterpart for larger v's, i.e., $v > 4$ in w-meshes and $v \geq 4$ in f-meshes. The performance in $\bar{t}$ of LRBF and SRBF policies with SRBP multiplexing is very close to FIFO-SRBP and hence is not plotted. LRBP multiplexing is less effective overall than SRBP: it has virtually no improvement in $\hat{t}$ over the CTS-only counterpart. But it still reduces $\bar{t}$ effectively when $v > 8$ for both w- and f- meshes, though not to the extent of SRBP multiplexing. From the data shown above for SRBP and LRBP multiplexing, one can conclude that they are quite effective in reducing the variance of message latencies, though the makespan is sacrificed somewhat for large v's.

We also ran simulations for the general case that message length is not uniform. Results are found to be consistent with the uniform message-length case, and bandwidth-sensitive message scheduling and flit-multiplexing methods are found to be more effective when the variance of message length is increased.

The simulation results in this chapter are summarized as follows.

- SRBF message scheduling outperforms LRBF in almost all situations. This is surprising since in Chapter 2, LRBF scheduling is shown to be much more effective than SRBF in a network with large-buffer switching methods like store-and-forward and virtual cut-through. We can thus conclude that in a wormhole-switched network, resource management should be quite different from large-buffer switching networks.

- Demand-driven allocation and CTS lookahead are very effective in reducing the waste of physical bandwidth, especially when the number of virtual channels is large.

- If reducing the mean latency is the main goal, then priority-based multiplexing is most effective. Especially, in the case of f-meshes with a large number of virtual channels, no other message-scheduling policy or flit-multiplexing methods can stop the trend of increasing $\bar{t}$'s with larger v's. With SRBP multiplexing, $\hat{t}$ is also reduced when v is small. Greedy multiplexing can be used as a quick-and-dirty alternative to reduce $\bar{t}$, but it is much less effective and may degrade $\hat{t}$ when v is large.

- When $v = 1$ or $v = 2$, f-meshes should be considered a better topology than w-meshes. For the case $v = 2$, f-meshes outperforms w-meshes in both $\hat{t}$ and $\bar{t}$ with less resources. Moreover, w-meshes cannot function with $v = 1$ unless they are used as f-meshes.

- Reducing the makespan of a mission does not necessarily reduce $\bar{t}$, and vice versa. Network configurations should be evaluated carefully with both measures before making any conclusion on their performance.

CHAPTER 4

MAPPING OF COMMUNICATING TASK MODULES IN HYPERCUBE MULTICOMPUTERS WITH POSSIBLE LINK FAILURES

4.1 Introduction

While the abundance of nodes in a hypercube multicomputer allows for executing tasks that require a large number of nodes, inter-node communication is still a major bottleneck in achieving the overall speedup [25, 66, 42]. To achieve communication efficiency, during the task execution, efforts must be made to improve traffic flow-control mechanisms as we have done in Chapters 2 and 3. These mechanisms are basically concerned with on-line, system-level implementation. Communication efficiency must also be improved on a per-task basis by exploiting the communication locality among task modules *before* the execution.

To map task modules for an "optimal" performance, the run-time behavior of these modules must be known *a priori* to some extent. However, as stated in the Halting Problem [76] in computing theory, there is no way to predict the exact run-time behavior of a program before it is actually executed. In case of distributed computation, it is also very difficult to predict the timing of communication events before a set of task modules are actually executed. As shown in the survey by Norman and Thanisch [85], the problem of module mapping has been addressed by numerous researchers [57, 78, 79, 77, 21, 34], using a wide range of models and solution methods. In the graph-mapping approach (e.g., [55, 98]) the timing aspects of module communication are ignored, and a simple objective function is proposed for optimization. It is generally difficult to relate this objective to any of well-known performance measures, such as task execution time. By contrast, any more complicated approach (e.g., [79, 21]) requires a substantial amount of knowledge of the run-time behavior of task modules, which may not be available unless the task is tested thoroughly beforehand.

Our primary goal here is to optimize communication performance. We use a relatively simple cost function and verify (with simulations) that optimizing this function actually leads to better communication performance, especially for mapping concurrently communicating task modules. Focusing on communication performance differentiates our work from others' related to more generic aspects of task mapping. Taking a communication-oriented approach to the task mapping problem is hardly a limitation, many researchers [25, 66, 42] agree inter-node communication is of the utmost importance to the performance of any multicomputer system.

This chapter is organized as follows. In Section 4.2, we present the basic system model and assumptions used. Our problem is also formally stated there. In Section 4.3, the NP-hardness of minimizing the proposed cost function is stated first in order to justify the use of heuristic algorithms. Several heuristics are then used to find good suboptimal solutions. These heuristics are tested extensively for various inputs to assess the performance of the mappings obtained from them. We then simulate these algorithms to verify the actual performance of the mappings found by minimizing the proposed cost function. The effects of inaccuracy in describing the task behavior are also discussed there. Section 4.4 deals with the case where an alternative fault-tolerant routing algorithm is used. Some remarks are given in Section 4.5 about the effects of combining the mapping strategies with flow-control mechanisms discussed in Chapter 2.

4.2 Preliminaries

The communication volume between each pair of modules is expressed as the total length of messages exchanged between them. Inter-module communications are assumed to be accomplished via message passing. A message is routed from the source to the destination via a fault-free shortest path under circuit or message switching. When there is no faulty link, it simply degenerates into the e-cube routing algorithm. Since we are considering the case with heavy concurrent traffic, as we have shown in Chapter 2, virtual cut-through degenerates into message switching and will not be discussed here.

Since most existing hypercubes do not support a per-node multi-programming environment, it is assumed that at most one module is mapped to a node, i.e., the mapping between nodes and modules is one-to-one. For a task with M modules such that $2^{n-1} < M < 2^n$ for some integer n, one can add some "dummy" modules and make it a task with 2^n modules. So, we will henceforth assume $M = 2^n$ where n is the dimension of the target hypercube to execute the task, and thus, the mapping of modules into nodes is one-to-one and onto.

For a network of nodes, we define a *communication event between modules* (CEBM) as an instance that a module needs to send a message to another module, while defining a *communication event between nodes* (CEBN) as an instance of a node needing to send a message to some other node. In circuit switching, these two are indistinguishable. In message switching, however, a single CEBM can become several CEBNs. For example, when a pair of modules reside in two different nodes which are two hops apart, in circuit switching a CEBM from one module to the other is just a CEBN from one node to the other node. For message switching, however, this CEBM becomes two CEBNs: one from the source to the intermediate node, and the other from the intermediate node to the destination. We said there is an *outstanding* CEBN if a message is to be sent by a node. An outstanding CEBN is said to be *processed* if it is sent from the source node to a neighboring destination node. An outstanding CEBN may not be processed immediately due to the limited link resources available. A CEBN is said to be *blocked* if it is not processed immediately.

The performance of a mapping is eventually measured at the execution time by its *communication makespan*, or *makespan* for short, which is the time span from the the first CEBN becoming outstanding to all CEBNs being processed. As an illustrative example, in Fig. 4.1 we have a simple network of 4 nodes with 3 CEBMs. The status of each link during the execution under both circuit and message switching is shown in this figure. Note that the computation time needed is invariant among different mappings, since at most one module is mapped to each node. Therefore, in this case, communication makespan is the main source of difference in the completion time of a task. However, communication makespan cannot be easily described as a mathematical function, and its value depends on the timing of communication events, which even for the same mapping, can change from one execution to the next due to the complex interactions among task modules and/or processor nodes. Hence it is impractical to use a direct optimization method. However, as we shall see, for concurrently communicating task modules, minimizing the proposed cost function generally leads to minimization of makespan.

The *communication cost* in executing a set of task modules is defined as the sum of time units during which links are kept busy with the messages among these modules. In other words, it is a measure of the total communication resources used by an instance of task

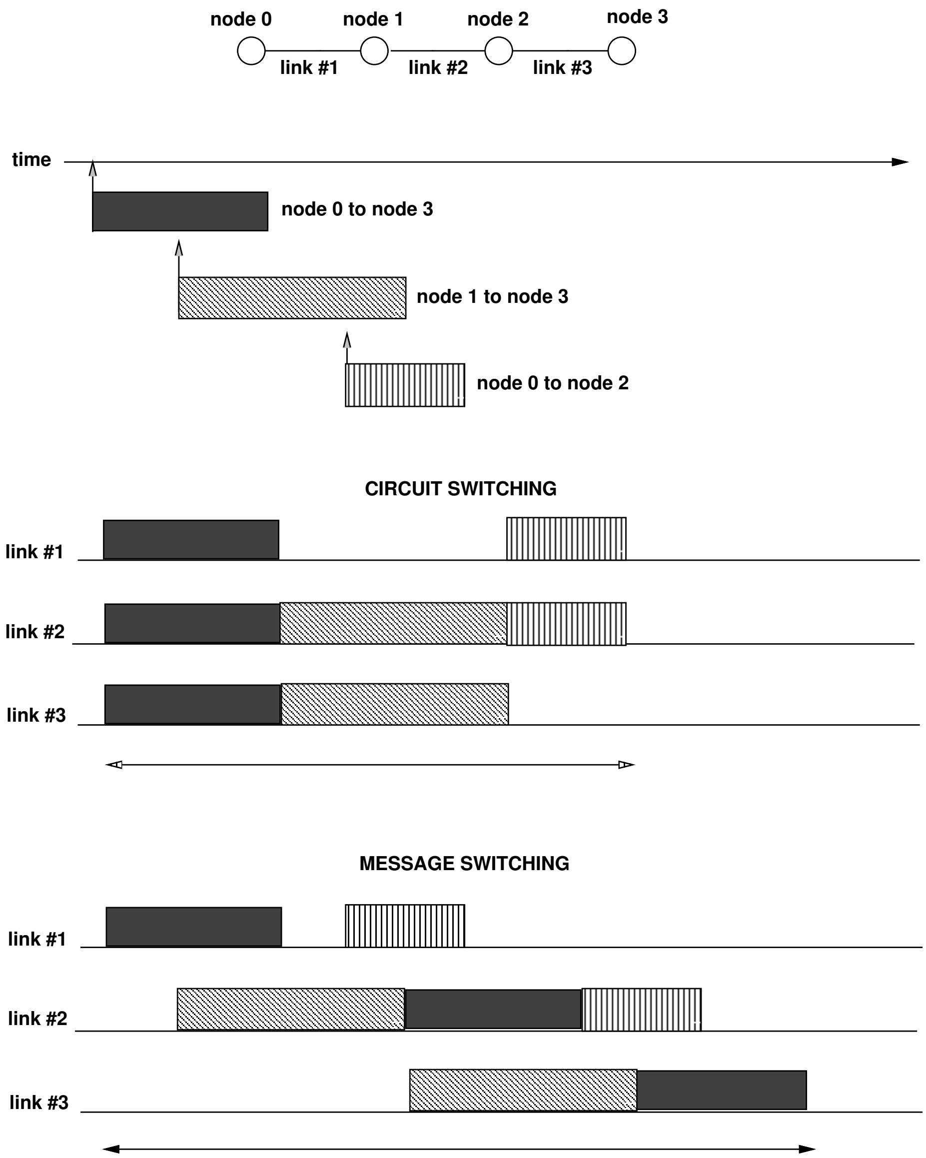


Figure 4.1: An example demonstrating the definition of makespan.

execution measured in time units. Suppose $c(h)$ is the number of time units links are kept busy with a unit-length message sent over a path of h hops. The sum of time units that links are kept busy for related purposes other than message transmission — such as establishing a connection — is assumed to be negligible. For message-switched hypercubes, $c(h) = hc(1)$, but this relation may not be accurate for circuit-switched hypercubes. However, if the "call request" signal to hunt for a free path occupies each link only for a very short time, then this expression would be a good approximation even for circuit-switched hypercubes. By defining $c(1)$ as a unit of communication bandwidth (i.e., the link usage by one unit-length message traversing one link), the communication bandwidth resulting from executing a task under a mapping Γ becomes: $k(\Gamma) = \sum_{1 \leq i \leq x} iV_i$, where V_i is the total length of messages traversing over i links, and x is the maximal path length in the hypercube. ($x = n$ if only shortest paths are considered.) One can easily see that $cost_{com}(\Gamma) \propto k(\Gamma)$.

In both type of switching, communication bandwidth is proportional to the total link occupation time, two communicating modules placed far apart will require more communication resources, and there is a higher possibility that some other instances of communication will be blocked and/or delayed, which can in turn lead to an increase of communication makespan. Therefore, we can predict that reduction of communication bandwidth is crucial to concurrently communicating tasks. When introducing the notion of communication cost and communication bandwidth, we deliberately avoided the low-level timing details. We only consider the total message length to be sent/received between a pair of task modules during the whole mission time, thus allowing for optimizing a simple cost function that can be translated into a combinatorial optimization problem.

The following notation will be used throughout this chapter:

- $D(n_i, n_j)$: the distance (i.e., the length of a shortest path) between node n_i and n_j, and is dependent upon the routing algorithm used. For now, we will assume $D(n_i, n_j) = D(n_j, n_i)$. (The case where $D(n_i, n_j)$ can be different from $D(n_j, n_i)$ will be discussed in Section 4.4.) Before mapping a module , $D(n_i, n_j)$'s are assumed given. Note that the distance between a pair of nodes may be greater than their Hamming distance and depends on the number of faulty links and the routing algorithm used.

- n: the dimension of the target hypercube.

- U: an $M \times M$ communication volume matrix, where U_{ij} is the communication volume from m_i to m_j during the task execution, and M is the number of task modules. As mentioned earlier, we will assume $M = 2^n$ unless specified otherwise. Note that $U_{ii} = 0 \; \forall i$, since a module does not send messages to itself.

4.3 Optimization Heuristics and Performance Evaluation

Although the cost function we proposed is simple in nature, optimizing it is an NP-hard problem. Its NP-hardness can be proved by restricting to the fault-free hypercube embedding problem discussed in [71]. Therefore, there exists no known polynomial-time algorithm to find an optimal mapping. Note that minimizing makespan, rather than communication bandwidth itself, is our ultimate goal. As we shall see, good heuristic algorithms will suffice in most situations. As shown in [107], an optimal solution that minimizes communication bandwidth is usually computationally expensive, and may only improve slightly over fast algorithms in terms of minimizing makespan, our actual objective.

One simple greedy heuristic which has been tested to work well in fault-free cases [107] is given below. Consider each task as a weighted graph with vertices representing modules and edge weights representing communication volumes. For any two nodes x and y under the e-cube routing, $D(x, y) = D(y, x)$. Therefore, it is sufficient to use an undirected graph with $U_{ij} + U_{ji}$ as the weight on the edge connecting m_i and m_j. We want to find a Hamiltonian cycle in this task graph with as high a total edge-weight as possible, and then embed this cycle into a Hamiltonian cycle in the hypercube. A Hamiltonian cycle in a fault-free hypercube can be easily found with Gray-code enumeration. In an injured hypercube with faulty links, however, there may not be any Hamiltonian cycle available for embedding. So, we define a *weighted relaxed* (WR) Hamiltonian cycle in an injured hypercube (with no disconnected node) as a relaxed version of Hamiltonian cycle, such that two nodes x and y can be linked in the cycle via a *virtual edge* which may be a path from x to y through some intermediate nodes. The weight on each virtual edge of the cycle is the number of physical edges on it. The greedy algorithm embeds the Hamiltonian cycle in the task graph with the maximum weight (found by a greedy approach) into the minimum-weight WR Hamiltonian cycle in the injured hypercube (also found via a greedy approach).

Two other (more complex) heuristic algorithms are also implemented and tested: a bottom-up approach algorithm based on the one proposed in [55], and a top-down approach proposed in [35]. Both of these algorithms are modified to handle cases with broken links. A third non-deterministic approach using the simulated annealing method [72] is also implemented and tested, where 2-opting [2] is used as the perturb function. To compare the quality of the mappings found by these algorithms with respect to communication bandwidth, we simulated these algorithms using input tasks with randomly generated communication volumes among their modules.

Each algorithm was executed for 10,000 randomly-generated tasks where U_{ij}'s are characterized by a normally-distributed random variable with mean μ and variance σ^2. Changing the value of μ is found to have little effect on the relative performance of mappings found with different algorithms as long as the ratio σ/μ remains constant. It is also found that, as σ/μ approaches zero, the difference in communication bandwidth between random mappings and those mappings found with the above three algorithms gets smaller, while the difference gets larger as σ/μ increases. This is consistent with the fact that when all U_{ij}'s are identical, all mappings will lead to an identical communication bandwidth, and all mappings algorithms will perform identically. For each data point, deviation from the mean value is found to be reasonably small ($< 3\%$).

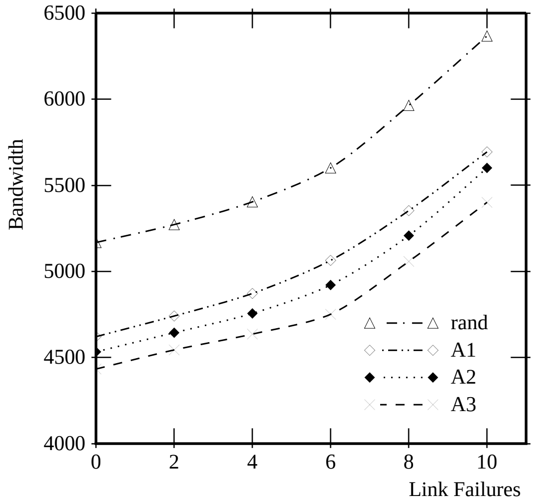


Figure 4.2: Performance of various algorithms.

For the input tasks used to obtain the plots in Fig. 4.2, U_{ij}'s are characterized with $\mu = 20$ and $\sigma = 15$; the horizontal axis depicts the number of faulty links while the vertical axis represents communication bandwidth. In this figure, "A1" represents the greedy algorithm, while "A2" represents the communication bandwidth achieved with either top-down or bottom-up algorithms, whichever yields smaller communication bandwidth. This is to enhance the readability of the plots since the performance of the top-down and bottom-up algorithms turns out to be very close to each other.

It can be seen from the above result that the greedy approach performs surprisingly well. Complex (i.e., top-down and bottom-up) approaches outperform the simple greedy approach only by a small margin. Furthermore, as the number of faulty links increases, the gap between the two curves gets narrower. This can be explained by the fact that both the top-down and bottom-up approaches are best suited for fault-free (thus regular) hypercubes. For hypercubes with faulty links, the interconnection structure is no longer symmetric or regular. In such a case, the partitioning mechanism in the top-down approach and the combining mechanism in the bottom-up approach must use less accurate heuristic decisions, hence degrading the performance.

The simulated annealing approach ("A3"), on the other hand, has shown more consistent performances. Its advantages over other algorithms become more pronounced as the cube size and the number of link failures increase. Therefore, we can conclude that this approach is more adaptable to irregular structures.

In Table 4.1, we show the relative timings of various algorithms used. The algorithms are tested on a DEC 5000 workstation running Ultrix operating system. Though we have only shown the performance data for problem size of $n = 4, M = 16$, the relative performances of different algorithms are found to be consistent at least up to the problem size of $n = 8, M = 256$.

	CPU Time			
Size	Greedy	Top-Down	Bottom-Up	S-Annealing
$n = 3, M = 8$	.057	4.2	7.8	125.2
$n = 4, M = 16$	.178	15.3	28.3	433.6
$n = 5, M = 32$	1.215	172.4	297.6	2537.1

Table 4.1: Timing comparisons for various algorithms.

To demonstrate why minimizing the proposed cost function, i.e., communication bandwidth, can be effective, we also need to compare the makespans of those mappings found with different algorithms. Our simulation model for this purpose is described below.

Timing: A time unit is selected as the time required to send a unit-length message over a communication link between a pair of nodes.

Routing algorithm and mechanism:

- Link failures are detected before task mappings and execution. Each message is routed through a fault-free shortest path determined prior to the execution of this task. We assume there are no additional link failures during the execution of this task.
- Under message switching, the routing mechanism at an intermediate node on a path will take a certain amount of time to forward a message from one link to the next. We assume this time to be relatively small and absorbed into the length of the corresponding message.
- As in Chapter 2, the hypercube is equipped with a communication adapter like SPIDER [31] at each node. The routing controller can send or receive multiple messages from different links at the same time. Also, incoming buffering/queueing and outgoing message transmission are done in parallel for all switching methods.
- Each communication link is half-duplex, i.e., at any instant of time, only one message can be sent in either direction of a link.
- The propagation delay on a communication path is assumed to be negligible.

Task communication behavior:

- ΔT, given for each task, denotes the time window in which the CEBMs arrive. The arrival times of CEBMs are uniformly distributed in $[0, \Delta T]$. Hence, for a set of task modules, a larger ΔT represents the communication being less intense, while a smaller ΔT represents the communication traffic being highly concurrent.
- L_{msg} denotes the maximum message length. The communication volume between each pair of modules is randomly grouped into messages of lengths within $[1, L_{msg}]$.

Message scheduling and queueing: If a link is busy when it is to be used for transmitting an incoming message, the message is stored in a FIFO queue at the source end of the link. When more than one message requests the use of the same link at a time, one of them is randomly chosen to use the link. This selection procedure is repeated until all requests are honored.

The goal of our simulation is to comparatively evaluate the performance of different mappings under the same execution environment, but not to compare the performance of different system implementations. So, the simulation results should not be used to determine the relative performance of different switching methods or routing algorithms.

The mappings found are fed into an event-driven simulator similar to the one used in Chapter 2 to evaluate their performance in a close to real-world environment. The results are plotted in Fig. 4.3 for message switching systems. Input tasks used here are the same as those used for Fig. 4.2. We set $L_{msg} \in [1,5]$, and $\Delta T = 100$. Results for circuit-switched hypercubes are found to be similar in most situations and thus are not presented.

The effects of changing ΔT under the same mappings for a given task are shown in Table 4.2 for message switching without link failures. The results are found to be similar to those under circuit switching. For the cases of $n = 4, M = 16$ and $n = 5, M = 32$, changing ΔT in the range $[10, 300]$ does not have any significant impact on the relative performance of mappings found with different heuristics. The mappings found with all of the above heuristics have shown substantial improvements over random mappings $\forall\ \Delta T \in [10, 300]$. This is because the network gets saturated with messages when $\Delta T = 300$.

	$n = 3, M = 8$				$n = 4, M = 16$				$n = 5, M = 32$			
ΔT	rand	A1	A2	A3	rand	A1	A2	A3	rand	A1	A2	A3
10	220	176	176	173	767	652	635	616	2456	2007	1994	1924
25	220	176	175	173	767	653	636	616	2469	2011	1995	1928
50	221	177	177	175	769	655	636	617	2474	2026	2005	1940
100	222	182	178	176	770	655	637	618	2475	2028	2013	1949
200	308	306	305	302	772	657	639	620	2476	2037	2025	1961
300	311	308	305	303	775	661	642	622	2478	2048	2031	1993

Table 4.2: Effects of changing ΔT under message switching.

In case of $n = 3, M = 8$, the network becomes less congested at $\Delta T \approx 160$ and the differences of makespans among different mapping algorithms start to diminish. So, we can conclude that minimizing communication bandwidth yields a peak improvement when the task modules are communication-bound and the communication network may become highly congested during the execution of this task. For $n = 4, M = 16$, the ΔT value which results in small performance differences is approximately 750, while for $n = 5, M = 32$ it is about $2,250$. However, when ΔT is relatively small and the network is not near saturation, the difference in message queue length can be made smaller by using the mappings obtained from the minimization of communication bandwidth. Depending on system implementation, the performance of a node may also be influenced by the length of message queue it has to maintain.

The effects of changing L_{msg} are more subtle than changing ΔT. Generally, shorter message lengths result in better performances in circuit-switched hypercubes, while for message-switched hypercubes, changing the message length does not affect system performance notably if the overall communication bandwidth is fixed.

Our simulation results have indicated that different switching techniques do not matter much to system performance for highly concurrently communicating tasks. Circuit switching and virtual cut-through is found to have only a slightly better performance than message switching for the same mappings. However, as mentioned earlier, the actual performance will depend on system implementation, and thus, the simulation results should not be used to compare the effectiveness of the two switching methods.

When the number of faulty links grows within our preset range (i.e., less than one third of all links), makespan also increases. For smaller hypercubes, such as $n = 3$, introducing even one more faulty link can make a significant difference in performance. This effect gets more pronounced when the number of link failures becomes larger, as one can see in Fig. 4.3. As the cube size increases, there will be more fault-free links, hence making lesser impacts of a single link failure on system performance.

Though the proposed mappings strategy requires only minimal information of run-time task behaviors, we still need the communication matrix to map a task. It is obvious that unless the task has been fully tested and each message length is exactly calculated, the entries in the communication matrix cannot be absolutely accurate. To study the effects of an inaccurate communication matrix, we repeated the simulation for evaluating makespan while introducing uncertainties in the communication matrix. In Fig. 4.4, the input tasks are essentially the same as those in Fig. 4.3, but there is a maximum of 20% error in each U_{ij}, i.e., during an instance of actual task execution, the total length of messages exchanged between m_i and m_j is $U_{ij} \pm 0.2U_{ij}$. From Fig. 4.4, one can see that inaccuracies in U_{ij}'s affect communication performance, especially when the cube size and number of link failures are large. However, when the number of link failures is less than one sixth of all links, the overall performances of various mappings algorithms are still quite close to those in the case with exact U_{ij}'s.

4.4 An Alternative Routing Algorithm

Thus far, we have assumed that the hypercube is implemented with a routing algorithm which routes messages from the source to the destination via fixed, shortest paths deter-

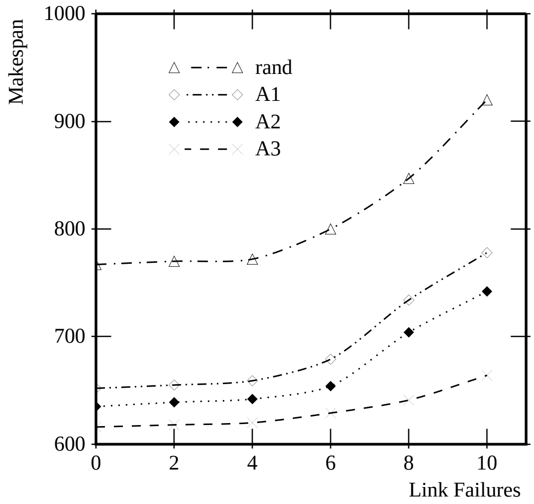


Figure 4.3: Performance of mappings under message switching.

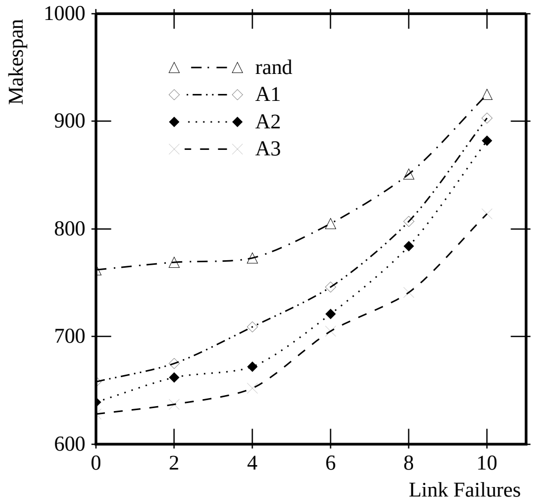


Figure 4.4: Performance of mappings under inaccurate task information.

mined before the execution of each task. However, there are several practical problems with this assumption. For instance, all faulty links must be known before the task execution, which may not always be possible. Also, if additional link failures occur later, the execution of the task may become unsuccessful.

To overcome these problems, we must use a routing algorithm that is more adaptive to system changes. For instance, the DFS algorithm proposed in [18] is an adaptive fault-tolerant routing algorithm which uses only a limited amount of global link status information. Under this algorithm, the system does not require *a priori* link status information, and communications can be completed even if some unexpected link failures occur during task execution as long as all nodes involved remain connected. However, due to the adaptive nature of the DFS algorithm, it is difficult to predict the length of the path used for routing a message during task execution, especially in the presence of link failures. So, $D(x, y)$ cannot be accurately calculated, thus making it difficult to decide the communication bandwidth of a mapping. Furthermore, under some adaptive routing algorithm like the DFS algorithm, due to the lack of global link status information, the length of the path chosen for communication from node x to node y may not be the same as the one chosen for that from y to x. For example, suppose we have a 3-cube with three broken links, $00*$, $0*0$, and $*01$. Then the length of path chosen under the DFS algorithm from 000 to 111 is 3. But the path chosen to route messages from 111 to 000 is 111 $\rightarrow$110 $\rightarrow$001 $\rightarrow$101 $\rightarrow$001 $\rightarrow$110 $\rightarrow$010 $\rightarrow$011 $\rightarrow$001 $\rightarrow$000, which has a length of 9. The routing algorithms with this nature are said to be *asymmetric*. In most cases, a routing algorithm becomes asymmetric only in the presence of faulty components.

Based on the above observations, one may jump to a conclusion that there is no way to minimize the communication bandwidth of an mapping, and hence it will be impossible to improve communication efficiency by appropriately placing task modules. However, as our simulation results show below, use of the proposed cost function, even by mapping task modules to the nodes as if there were no faulty links, can still significantly improve communication performance over random mappings when the number of faulty links is within a certain range.

Three mapping strategies are compared in our simulation. The first is the usual random mapping. The second is to apply the greedy algorithm to the hypercube without knowing which links are faulty. The third assumes perfect knowledge of link failures and how each message will be routed during the execution. This strategy is an unrealistic, ideal case, which gives an upper bound of performance improvement with communication bandwidth, whereas the second strategy provides a lower bound. In real applications, depending on the knowledge available during the task mapping phase, the performance should lie somewhere between these two extremes.

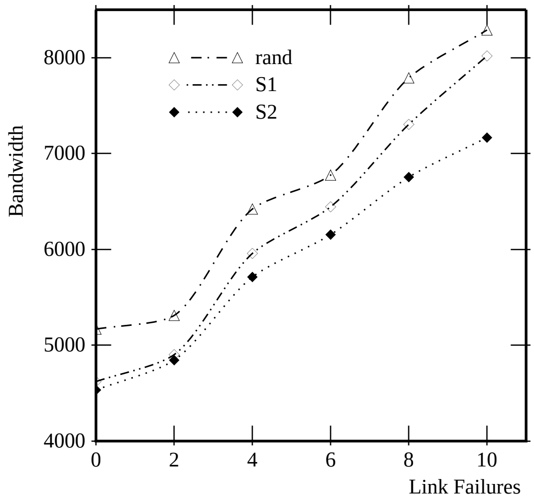


Figure 4.5: Communication bandwidth under the DFS algorithm.

Fig. 4.5 shows the communication bandwidth of the mappings under the DFS algorithm for the same set of input tasks as in Fig. 4.2. “S1” represents the mappings found with no knowledge of faulty links, while “S2” represents those found with complete knowledge of faulty links and the routing paths of all messages. It can be easily seen that under the DFS algorithm, the overall communication bandwidth is higher than the routing algorithm used before. Nevertheless, the mappings “S1” still generate smaller communication bandwidth than random mappings, though the improvement becomes insignificant as the number of faulty links increases. The same set of input tasks used in Fig. 4.3 are employed again for event-driven simulations, except that the DFS algorithm is used here. Since the DFS algorithm is designed based on the operating principles of message switching, we only simulate the hypercubes implemented with this switching method.

The measured makespans of these mappings are plotted in Fig. 4.6. It is found that, without knowledge of faulty links, mappings “S1” still improves over random mappings with a margin of at least 10% when the number of faulty links is more than one eighth of the total links. This margin increases as the number of faulty links increases, but starts to level off when the percentage of faulty links approaches 33%. The mappings “S2” show even larger improvements and improve over random mappings with a steadily increasing margin as the number of link failures increases.

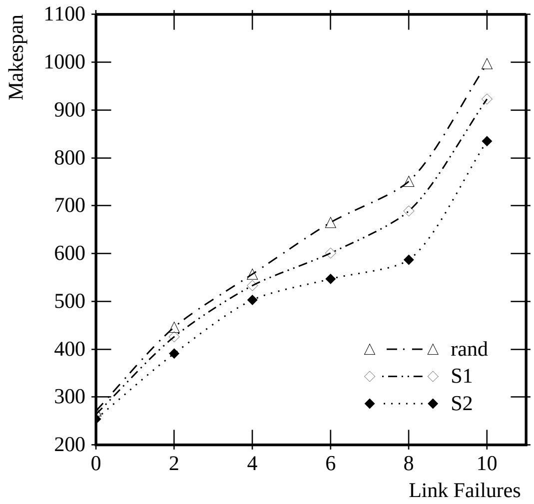


Figure 4.6: Performance of mappings under the DFS algorithm.

By comparing Fig. 4.6 with Fig. 4.3, one can see that, though the DFS algorithm results in an overall higher communication bandwidth, it results in smaller makespans when the number of faulty links is relatively small. This is due to the fact that the DFS algorithm chooses communication paths in a more "spread out" fashion and causes less congestion than the shortest fixed-path algorithm used before. This advantage diminishes after the number of faulty links grows beyond one fifth of all links. When the percentage of faulty links reaches 25%, the DFS algorithm begins to yield larger makespans than the shortest path routing. This is because paths available between nodes are becoming fewer, so messages cannot be spread out to more paths under the DFS algorithm. Also, since the DFS algorithm does not always route messages through a shortest path, the greater communication bandwidth overhead of the DFS algorithm starts to have dominant effects. Note, however, that implementation details will be crucial in actual applications and these simulation results should not be used to judge the relative merits of different routing algorithms.

4.5 Remarks

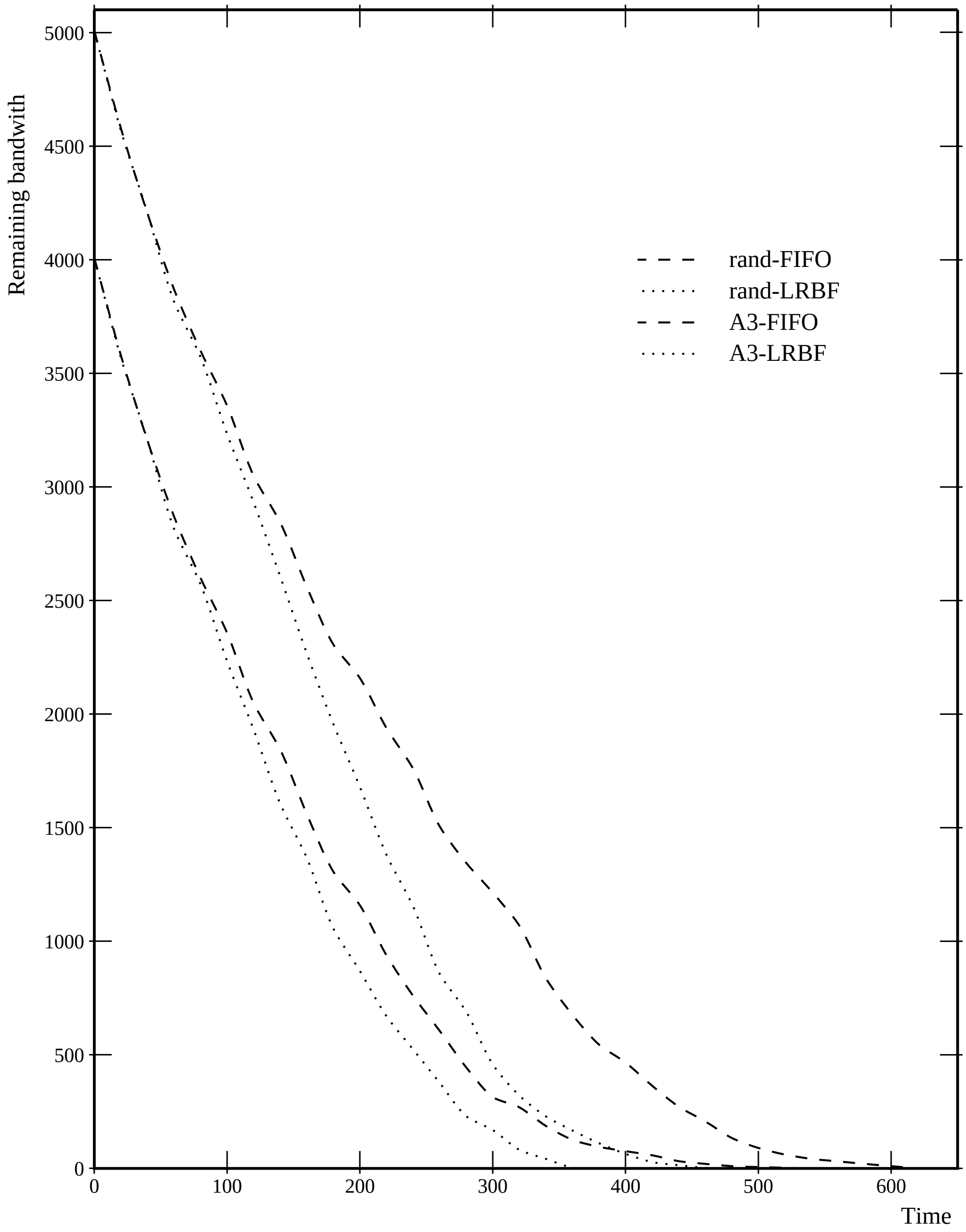


Figure 4.7: Plot of remaining bandwidth versus time.

In Chapter 2, we have shown that by employing appropriate message scheduling policies, the performance of a binary hypercube under concurrent communication traffic can be greatly improved. Here we will demonstrate that by applying these run-time flow-control mechanisms, the communication performance of mappings optimized with respect to communication bandwidth can be further improved.

In Fig. 4.7, we demonstrate the effect of applying a message scheduling policy on the performance of a mapping. The performance measurements are shown for a typical input task used in previous simulatios for the case $n = 4, M = 16$ and $\Delta T = 0$, and the network coonsidered here is fault-free. The *remaining bandwidth* during executions is plotted against time for random mapping and the mapping optimized with respect to communication bandwidth using the simulated annealing method ("A3"). The mappings are executed on systems with the FIFO message scheduling policy and the LRBF policy defined in Chapter 2. "LRBF" denotes the message scheduling policy which gives a higher priority to the message with the largest remaining bandwidth.

It is obvious that the message scheduling policy can further improve the performance of mappings. Note that given a mapping, different flow-control mechanisms result in different rate of "energy"(remaining bandwidth) dissipation. Better flow-control not only results in a higher rate, but also a more linear behavior in the curve, and hence more predictable task communication response time. On the other hand, an optimized mapping leads to lower "initial energy", and reduces the time needed to dissipate it. Note that the mapping strategy can work almost independently of the flow-control mechanisms. And their improvements on the performance can be additive. The result also confirms that, the amount of initial commnuication bandwidth is a good indication of the quality of mappings.

CHAPTER 5

MAPPING CONCURRENTLY COMMUNICATING MODULES ONTO MESH MULTICOMPUTERS EQUIPPED WITH VIRTUAL CHANNELS

5.1 Introduction

The author of [85] provided a thorough survey on models and techniques used for mapping task modules in multicomputer systems. Among the various models, the graph-theoretic model has been widely used [55, 98, 94]. In this type of model, a set of communicating task modules is modeled as a *task graph*, and the interconnecting topology of processor nodes is modeled as a *processor graph*. An objective function is chosen for optimization when the task graph is embedded into the processor graph.

Using the graph-theoretic model, the mapping problem is conceptually equivalent to a combinatorial optimization problem, and there are a wide range of existing optimization algorithms [13] to solve it. However, since graph-theoretic formulation often oversimplifies the real-world situation, the objective function may not reflect any true performance measure. As a result, finding an "optimal" mapping with respect to the objective function may be meaningless. Here we basically adopt the graph-theoretic model, but our approach differs from the previous work as follows.

1. We focus on optimizing the communication performance, particularly the concurrent communication efficiency, of a mapping into a virtual-channel network.

2. Instead of using an abstract objective function, we adopt a realistic performance objective as the goal of optimization.

3. The unavoidable inaccuracies in predicting run-time task behaviors are taken into account.

The network under consideration employs wormhole switching. Each pair of adjacent nodes are connected by a pair of unidirectional physical links/channels. A fixed number of uni-directional virtual channels are time-multiplexed over each physical channel. Though most of our discussion may apply to general networks, we will focus primarily on the mesh network topology, especially k-ary 2-cubes which have been widely used in evaluating the performance of virtual-channel networks [25, 26].

A set of task modules are said to be communicating *concurrently* if they send messages to one another within a short span of time. A network under concurrent traffic arrivals is analogous to a physical system with a certain amount of injected *energy*, i.e., the total communication bandwidth. The time span required to *dissipate* the injected energy indicates the capability of the network to handle transient loads which is similar to the transient response of an electronic component to a step-function input. Given the same amount of initial bandwidth, i.e., initial energy, various flow-control mechanisms result in different *rates* of energy dissipation, and hence different time needed to dissipate all the energy. As shown in Fig. 5.1, the *remaining bandwidth* is plotted against time for two different flow-control mechanisms. Obviously, the time required to dissipate all the energy also depends upon the amount of the initial energy. Therefore, if the communicating modules in the network can be mapped so that the initial energy is minimized, i.e., communication locality is maximized, then the performance can also be optimized.

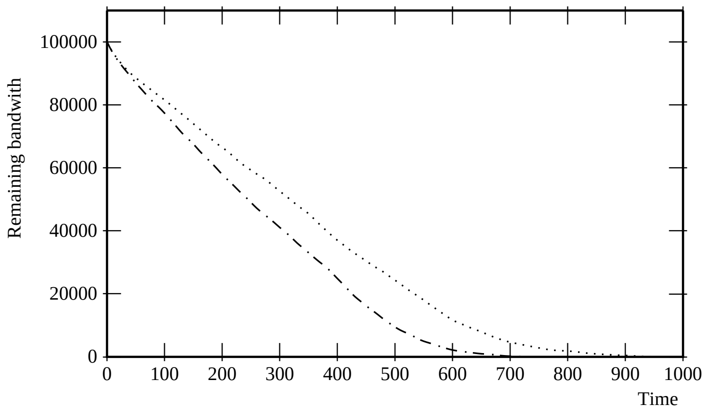


Figure 5.1: The remaining bandwidth versus time under two different message scheduling policies.

In Chapter 3, we have addressed the problem of run-time concurrent traffic flow-control.

Here we will discuss the issues of mapping concurrently communicating modules into a mesh multicomputer *before* run-time, so that at run-time, the communication performance can be optimized. Particularly, we will focus on the case where a substantial number of messages can be transmitted through the network simultaneously, and thus may cause serious traffic congestion. As pointed out in [25], most algorithms are communication rather than processing limited. Fine-grain parallel programs can execute less than ten instructions in response to an inter-processor communication event. To execute such programs efficiently, the communication network must be able to allow a large fraction of the nodes to transmit messages simultaneously.

The delivery of these concurrently-transmitted messages may not be mutually independent. The arrivals of all messages at their destination as a whole are often important for continuing the task execution. For example, in the execution of parallel algorithms, such as parallel state-space-search [92, 4], parallel sorting [104] and parallel Fourier-Transform [29], at certain stages, modules on each node must receive partial results contained in the messages from other nodes to continue. Therefore, in addition to the usual latency measurement, the *makespan* of a set of concurrently-sent messages will also be used for performance evaluation. The makespan of a set of messages is defined as the time span from the arrival [1] of the first message until all the messages reach their destination.

Our objective is to map modules into the host multicomputer to minimize makespans and latencies of those concurrently-sent messages. As we will show later, due to the complex interactions among messages in a virtual channel network, a direct optimization approach to the objective is extremely difficult. We will adopt an *indirect* approach by optimizing a simplified cost function, which can also lead to optimized performance with respect to the real objective. Several cost functions are proposed and then evaluated using simulations.

This chapter is organized as follows. Basic terms and concepts necessary for our discussion are defined in Section 5.2. Simulation results are presented and discussed in Section 5.3.

5.2 Preliminaries

A k-ary n-cube consists of k^n nodes arranged in an n-dimensional grid. Each node is connected to its Cartesian neighbors in the grid. A 2-dimensional $k \times k$ flat mesh is a subgraph of a k-ary 2-cube, is not a regular graph, and has less edges than the corresponding

[1] Here the *arrival* of a message means it is ready to be sent out of the source node.

k-ary 2-cubes (no wrap links at its boundary nodes). For convenience, we will call a k-ary 2-cube a *wrapped mesh*, or a *w-mesh* for short. Likewise, we will call a 2-dimensional flat mesh an *f-mesh*. Since an f-mesh is a subgraph of w-mesh with the same number of nodes, a w-mesh can also be made to function as an f-mesh by not using its wrap links.

Flow control in a virtual-channel network is performed at three levels: routing algorithms, message-scheduling policies, and flit-multiplexing methods. Each of these can be implemented with a variety of algorithms.

Routing Algorithms: Selection of paths for messages. We consider only non-adaptive routing. A message is routed to its destination via a fixed, shortest path. Issues related to fault-tolerance are not considered, or physical and virtual channels are assumed to be fault-free. In f-meshes, e-cube routing is used. The address of each node is expressed in terms of X and Y coordinates. A message is routed first in the X-direction until the Y coordinate of the node matches that of its destination node. It is then routed in the Y-direction. In w-meshes, a modified version of e-cube routing is implemented to utilize the extra communication links so that each message is routed via a shortest path. Deadlock-freedom is ensured by using the scheme proposed in [24]. That is, the virtual channels corresponding to each uni-directional physical channel are divided into high and low channels. Routing restrictions are then imposed such that either a high channel or a low channel, but not both, is allocated to each given message. The w-meshes need at least two virtual channels per physical channel to achieve deadlock-freedom.

Message-Scheduling Policies: Determining which message is allowed to access a free virtual channel in case of contention. When the number of messages to access a physical channel at the same time is larger than the number of available virtual channels, some of these messages have to be queued. So, we need to determine which messages are allowed to access the virtual channels, and which messages to be queued. When evaluating cost functions, we will mainly use the FIFO policy as a default, but other scheduling policies will also be tested when necessary.

Flit-Multiplexing Methods: Determining the way messages are time-multiplexed over a physical channel. When there are multiple virtual channels per physical channel, the messages allocated to these virtual channels are multiplexed over the physical channel. Flit multiplexing determines the order for these flits from different virtual channels to access the physical channel. In the *round-robin (RR)* multiplexing, virtual channels take turns in accessing the physical channel without using any network or message information. RR multiplexing without any modification will henceforth be called *strict* RR. *Demand-driven (DD)* allocation can be used to rectify the problem of wasted physical bandwidth in strict RR. With DD allocation, virtual channels will contend for use of a physical channel only if they have flits to send. Furthermore, with *CTS (Clear-To-Send) lookahead*, virtual channels only contend for use of a physical channel if each of them has a flit to send *and* the receiving node has room for it. This can further reduce the waste of physical bandwidth. Like message sequencing, flit multiplexing can also be priority-based, as discussed in Chapter 3.

We summarize the notation used in our discussion as follows.

- a_i^s: the address of the source node of message m_i.
- a_i^d: the address of the destination node of m_i.
- C_{xy}: the maximum number of messages that share a physical channel from node x to y.
- $d(x, y)$: the length of the shortest path from node x to y.
- F_{xy}: the maximum number of flits that share a physical channel from node x to y.
- ℓ_i: the length of m_i in number of flits.
- L_i: the set of physical channels in the path of m_i.
- m_i: a message in P.
- P: the set of messages to be sent concurrently among modules.
- $\mid P \mid$: the number of messages in P.
- t_i^a: the arrival time of m_i.
- t_i^0: the time span from the moment when p_i is ready to be sent till its header flit arrives at the destination.
- t_i^q: the accumulated queueing delay of m_i's header flit from the source to the destination.
- t_i^x: the accumulated multiplexing delay of m_i's header flit from the source to the destination.

- t_i^l: the latency of m_i.
- $\tilde{t}_i^l$: the estimated latency of m_i.
- ΔT: the size of the time window where messages in P arrive.
- v: the number of virtual channels per physical channel.
- $< x, y >$: a physical channel from node x to y.

Our discussion will be based on the following assumptions.

1. All mappings are one-to-one, i.e., each processor can at most be assigned one module.
2. Accurate values of ℓ_i's (message lengths in flits) are given. The effects of inaccurate ℓ_i's will be addressed later.
3. A physical channel takes one unit of time to transmit a single flit. A unit of time is also called a physical-channel *cycle*.
4. There is a single-flit buffer associated with each virtual channel.
5. A message arriving at its destination is consumed without waiting.
6. There are an even number of virtual channels associated with each physical channel in a w-mesh.

Problem Statement: Given a set of modules and a set P of messages to be sent among them, we want to map these modules into the network so that the makespan and average latency to deliver all the messages in P are minimized.

To select a mapping out of a large number of possible mappings, there must be a certain function to determine the quality of mapping. The most obvious choice is using the performance objective itself. In our case, the average latency of messages in a set P can be expressed as

$$\sum_{m_i \in P} t_i^l / \mid P \mid,$$

and their makespan can be expressed as

$$\max_{m_i \in P}(t_i^a + t_i^l).$$

In these equations, t_i^l can be expressed as

$$t_i^l = t_i^0 + (1/r_i)(\ell_i - 1).$$

The first term, t_i^0, denotes the time span between the arrival of m_i at the source node and the arrival of its header flit at the destination node. t_i^0 is composed of two components: the accumulated queueing delay t_i^q and the accumulated header flit multiplexing delay t_i^x. t_i^q is the sum of queueing times at all nodes in the path waiting for an available virtual channel. t_i^x is the sum of times m_i's header flit waits at the output buffers of nodes on its path for use of physical channels. The second term, $(1/r_i)(\ell_i - 1)$, represents the time required for all other flits of m_i to arrive at the destination, which is determined by ℓ_i and the transmission rate, r_i, of the pipeline set up for m_i. Depending on the flit-multiplexing method used and network condition, r_i may change with time during the transmission of m_i. Also, given a set of messages and fixed v, t_i^q will be affected by the underlying message-sequencing scheme. Therefore, even when the exact values of ℓ_i's are given, it is still very difficult to predict t_i^l's. Consider the simplest case of f-meshes using strict RR multiplexing without DD allocation or CTS lookahead. We have $r_i = 1/v \; \forall i$, hence only t_i^0 needs to be calculated.

- Suppose $v \geq C_{xy} \; \forall x, y$, i.e., no messages will be blocked, then $t_i^q = 0$, and $t_i^0 = t_i^x$. However, $t_i^x \in [0, \; v * d(a_i^s, a_i^d)]$, and as v and $d(a_i^s, a_i^d)$ become larger, it is more difficult to predict t_i^x's.
- When the number of concurrent messages is large and blocking is inevitable, t_i^q is no longer 0, and the value of t_i^0 is even less predictable.

With DD allocation or CTS lookahead, r_i is no longer a constant. It becomes even more complex in the case of w-meshes with partitioning of virtual channels for deadlock-avoidance. We therefore conclude that a direct optimization on the performance objective itself is not practical. We need to come up with a certain simplified function that, when mappings are optimized with respect to it, the resulting performance is also optimized. In this chapter, we will investigate low-complexity cost functions whose computational complexities are of the order $\theta(\mid P \mid)$.

A message $m_i \in P$ is characterized by its source and destination modules, ℓ_i, and t_i^a. Of these parameters, the message arrival time, t_i^a, is the most difficult to obtain accurately beforehand, since t_i^a will be affected by the precedence relationship among modules, and is intrinsically hard for a compiler or loader to analyze before actual execution. Even if the modules can be test-executed, message arrival times can still vary among executions of the same set of modules due to minor variations in the environment, such as clock-frequency drift. Besides, introducing time-related parameters into the optimization process can further complicate the problem by adding the scheduling aspect into the picture. Since we are mostly interested in dealing with concurrently-communicating modules, ΔT should be relatively small. We therefore propose cost functions which ignore ΔT and assume $t_i^a = 0$, $\forall m_i \in P$. Nevertheless, as we will show in our simulations, mappings optimized with a properly-chosen cost function still perform well when ΔT is large. Besides, the issues of message scheduling can be handled at run-time and, as we will demonstrate, can further improve the performance of a mapping.

The following cost functions will be evaluated:

- f_1: makespan estimate. The estimated value of t_i^l, denoted by $\tilde{t}_i^l$, is computed by

$$z + \min\{v, \max_{<x,y> \in L_i} C_{xy}\}(\ell_i - 1),$$

  where z is the estimated time required for the header flit of m_i to reach its destination. z is computed as $random() * \sum_{<x,y> \in L_i} F_{xy}$, where $random()$ is a random number uniformly distributed in $(0, 1)$. The estimated makespan is computed by taking the maximum of $\tilde{t}_i^l$'s. Since r_i assumes the lowest possible value, this will be a pessimistic estimate.

- f_2: average latency estimate. Similar to f_1, except that the average value of $\tilde{t}_i^l$'s is computed.

- f_3: sum of length-distance product, i.e., the total physical bandwidth required by the messages in P. Formally, it can be expressed as $\sum_{m_i \in P} \ell_i * d(a_i^s, a_i^d)$. This cost function is shown to be quite effective in large-buffer, non-multiplexing networks in Chapter 4. However, in a virtual channel network with wormhole switching, apart from physical bandwidth, the usage of buffer resource is also a major factor in network performance.

- f_4: $\max_{0 \leq x,y < M} C_{xy}$, the maximum number of packets to go through a physical channel, i.e., maximum congestion.

- f_5: $\sum_{0 \leq x,y < M} C_{xy}$, the sum of congestion on all physical channels.

- f_6: $\max_{m_i \in P}\{\sum_{<x,y> \in L_i} F_{xy}\}$, the maximum number of flits to go through a physical channel on the paths of all $m_i \in P$.

- f_7: $\sum_{m_i \in P} \{ \sum_{<x,y> \in L_i} F_{xy} \}$. Similar to f_6 but a summation is taken instead. Note that f_7 is different from f_3. In f_7, a flit can be counted several times if the physical channel it goes through are shared by a number of paths.

- $f_5 \mid f_3$: f_5 *constrained* by f_3, i.e., a mapping is considered better only if it has smaller values of f_5 *and* f_3.

- $f_7 \mid f_3$: f_7 constrained by f_3.

It is obvious that finding true optimal mappings with respect to each of the above cost functions is NP-hard [41], i.e., there are no known polynomial time algorithms. Also, finding optimal mappings with respect to them is not very meaningful since the cost functions themselves are not the actual performance objective. Therefore, our goal is to obtain good sub-optimal mappings with respect to a cost function with a reasonable computing time, and the mappings will perform well at run-time and show significant improvements over random mappings. We will adopt the simulating annealing method [72] for this purpose.

The advantages of using this method include:

- Computation times can be easily controlled by setting a few parameters, such as the initial temperature, freezing point, etc. Hence it is possible to evaluate cost functions by subjecting them to approximately the same amount of computing time.

- The performance of mappings may be continually improved with additional computing time.

- Possible parallelized implementation.

The termination of a simulated annealing process is generally decided by the following parameters: the initial temperature, the freezing point, the temperature updating function, and the exit criteria at each temperature. For each tested cost function, we carefully select these parameters so that for a given input traffic pattern and traffic density, the optimization process will terminate in approximately the same number of *trials*, n_T, regardless of the cost function used. A trial is defined to be an instance of randomly choosing two modules and exchanging their positions, followed by the evaluation of the cost function. On a Sun IPX workstation, the compiled C program requires approximately 20 seconds of CPU time for 500 trials. For each given n_T, only those inputs that the optimization process terminates after $n_T \pm 10\%$ trials are used. The resulting mappings are collected and their average performance is calculated. Note that we do not artificially force a simulated annealing process to stop. Instead, we choose the parameters carefully and discard inputs which can lead to early or late terminations when using any of the cost functions. By doing this, we can ensure the fairness when comparing the effectiveness of cost functions.

5.3 Performance Evaluation

The mappings optimized with respect to the various proposed cost functions are fed into the network simulation program we used in Chapter 3. Recall that we developed the simulator under the following assumptions:

- Transferring a flit between two nodes via a physical channel takes one unit of time.
- At any instant of time, all flits that have been allocated channels are transferred synchronously in a single physical channel cycle.
- Each virtual channel is assigned a single-flit buffer.

The simulation results presented here were obtained using the following parameters:

- FIFO is the default message-scheduling policy. The default flit-multiplexing method is RR with DD allocation.
- Unless stated otherwise, all messages are 20 flits long.
- There will be at most one message from one module to the other.
- Both w- and f- meshes are of size 16×16. Since performance trends are mostly similar for f-meshes and w-meshes for the same P, unless stated otherwise, only the data obtained with w-meshes are plotted.
- The number of communicating modules is fixed at 256, i.e., the same as the number of nodes in the network.
- The default number of virtual channels is $v = 4$.
- The Concurrent Communication Probability (CCP) that node i sends a message to node j in the uniform traffic pattern is 0.01. In a 16×16 network, the total number of concurrent messages during a mission is $\approx 0.01 \cdot (16^2 - 1)^2$.

- Unless stated otherwise, the traffic pattern is uniform. In hot-spot traffic, 5 hot spots in the network are randomly chosen, with CCP = 0.5 between any node and each of the hot spots.
- The default value of ΔT is 0.
- The modules are first randomly mapped.
- Each data point is obtained by averaging results from 10,000 mappings. Deviation from the mean values is found to be reasonably small ($< 5\%$).

In Figs. 5.2 and 5.3, the makespans of average latency of mappings optimized with different cost functions after ≈ 500 trials, (i.e., $n_T = 500$), are compared for different values of v. The performance of f_1 and f_2 are found to be very close to that of f_4 and f_6, and hence are not shown. f_1 and f_2 are only found to be effective in the case of f-meshes with large v's and small CCP values. This can be attributed to the fact that only in these situations makespan and latency estimates are more accurate.

From the results shown, it is obvious that f_7, $f_5 \mid f_3$ and $f_7 \mid f_3$ perform better than the other cost functions in this case. Mappings optimized with these functions are also more resilient to the change of v's. On the other hand, mappings optimized with some functions (e.g., f_4 and f_6) perform well with small v's but become worse with larger v's. In the case of f_4, mappings optimized with it improve over the random mappings when $v \leq 6$, but actually perform worse than random mappings when v gets larger. A similar behavior can also be observed from mappings optimized with the other mini-max type cost function, f_6, though to a less pronounced degree.

In Figs. 5.4 and 5.5, the performance of mappings optimized with the various cost functions under uniformly-distributed traffic are evaluated with variable n_T's. The number of virtual channels is fixed at $v = 4$. A good cost function should demonstrate a more predictable behavior, i.e., better performance measurements and less fluctuations when n_T is increased. Note that for each plotted curve, $n_T = 0$ corresponds to random mappings. Among the cost functions investigated, f_1, f_2, f_4 and f_6 all demonstrate highly unpredictable behaviors with increasing n_T. With more computing effort, mappings optimized with these cost functions can often worsen performance. This phenomenon is especially prominent with the makespan measurement. On the other hand, f_3, f_7, and those related functions like $f_5 \mid f_3$ and $f_7 \mid f_3$, all have more predictable performance, at least up to a much larger n_T than other cost functions. In makespan measurements, f_3 shows continual improvement of

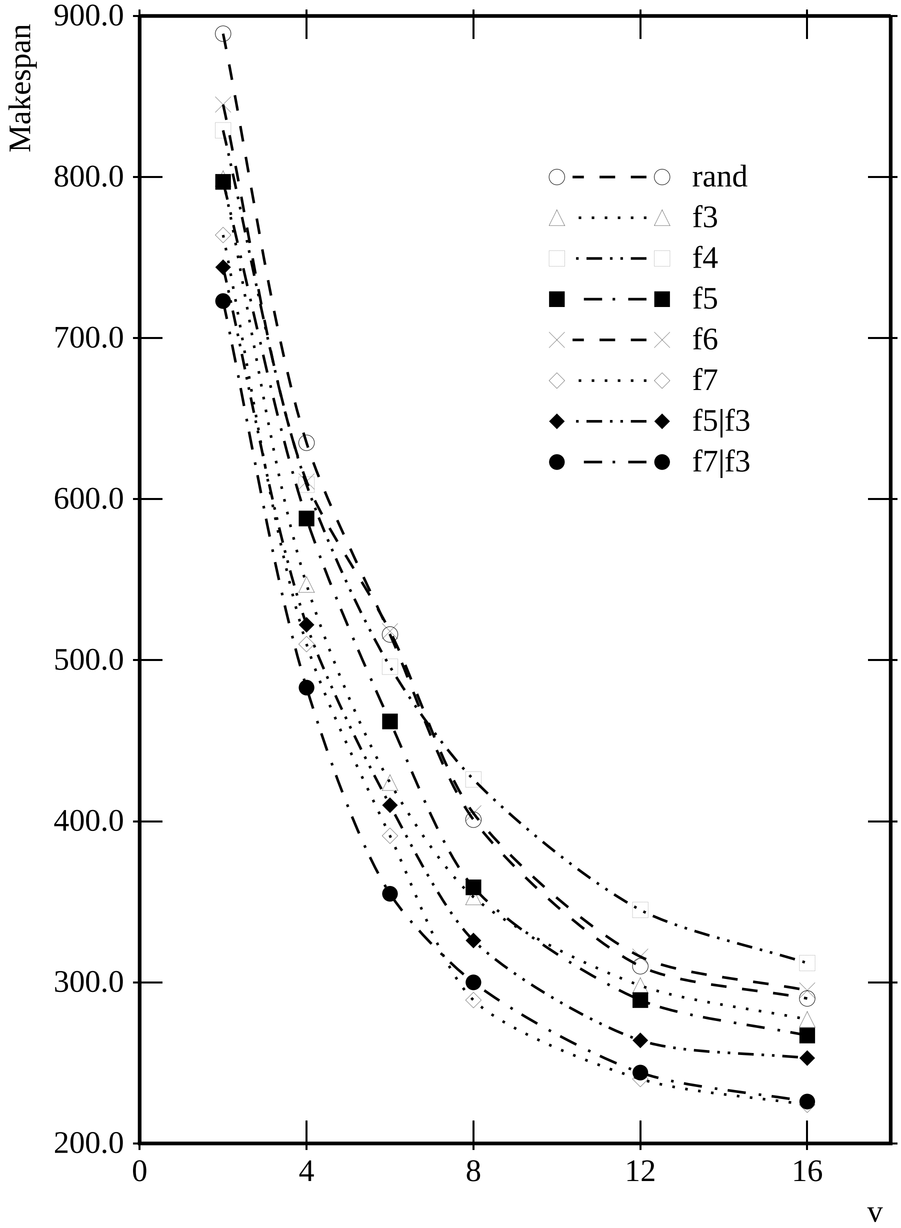


Figure 5.2: Makespan comparison of mappings optimized with various cost functions.

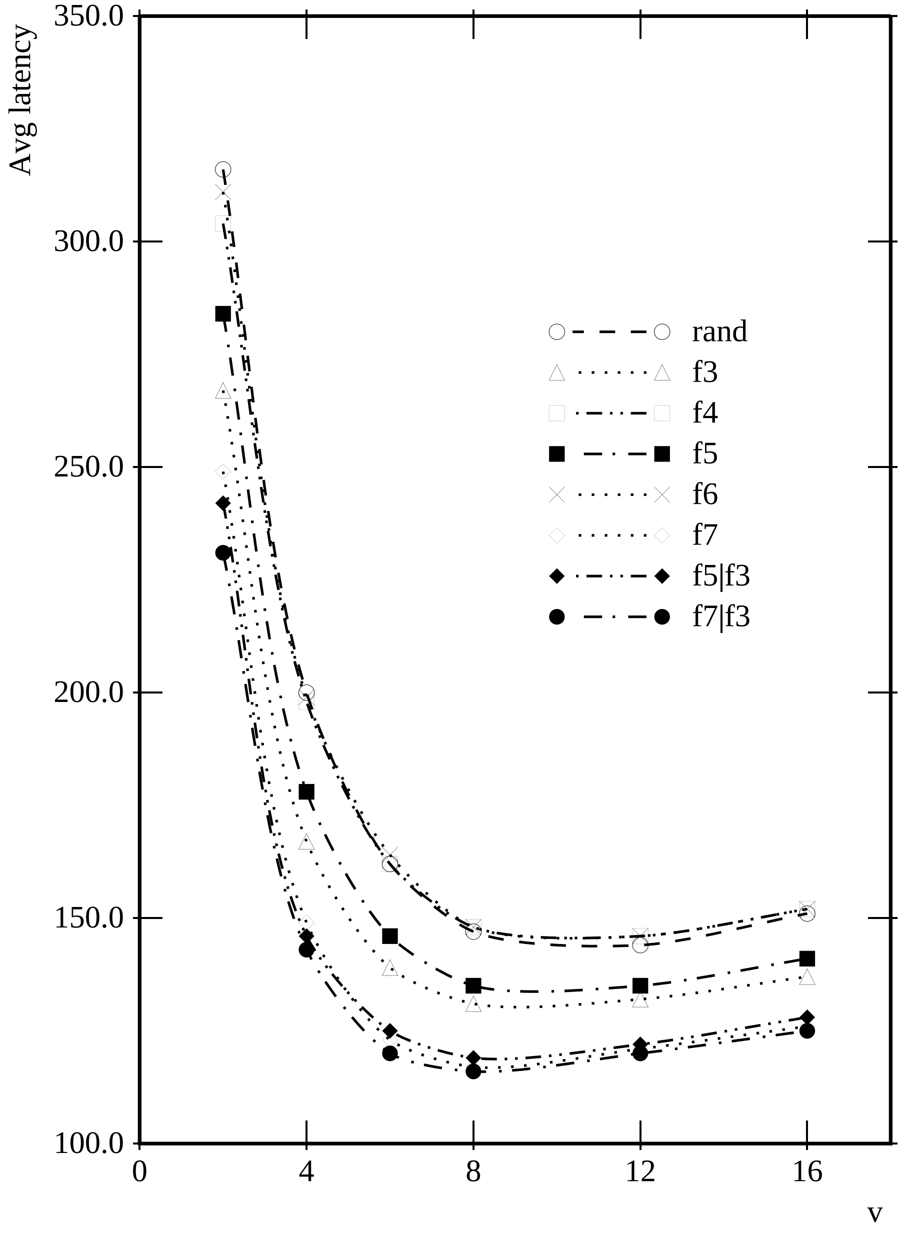


Figure 5.3: Average latency comparison of mappings optimized with various cost functions.

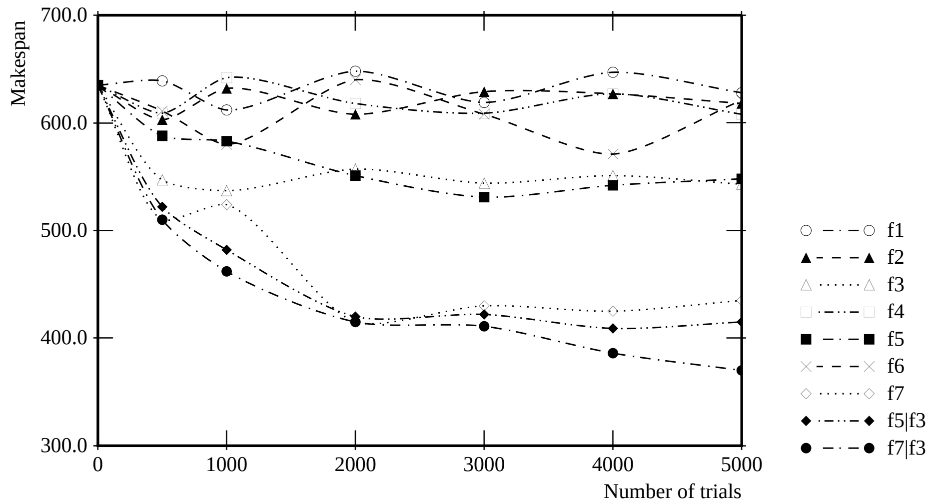


Figure 5.4: Makespan comparison of mappings optimized with various cost functions, uniform traffic.

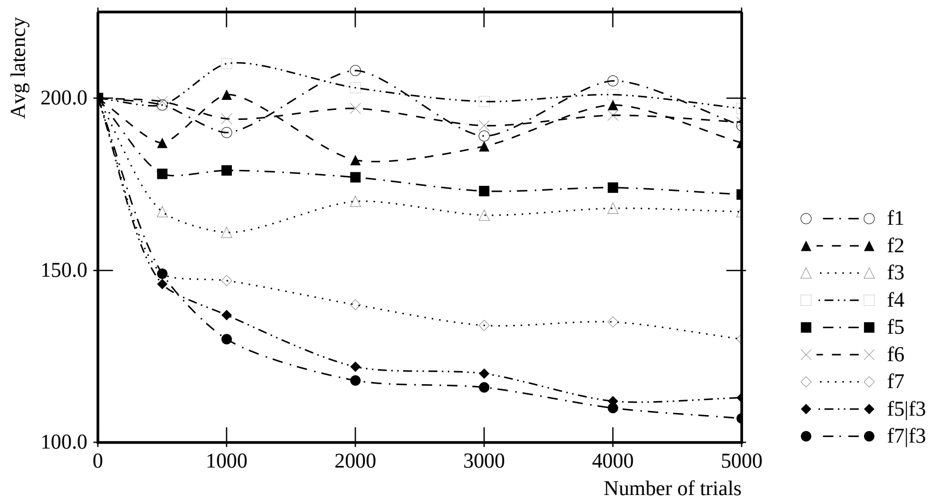


Figure 5.5: Average latency comparison of mappings optimized with various cost functions, uniform traffic.

mappings up to $n_t = 3000$, and $f_5 \mid f_3$ can improve up to $n_T = 4000$. While $f_7 \mid f_3$ is found to improve mappings continually up to $n_T = 10,000$. For average latency measurement, there are less fluctuations for all cost functions. However, mappings optimized with most cost functions stop making noticeable improvement after $n_T \geq 1000$. Only f_7, $f_5 \mid f_3$ and $f_7 \mid f_3$ show continual improvement for $n_T > 1000$, while $f_7 \mid f_3$ shows improvement even when $n_T > 8000$.

Figs. 5.6 and 5.7 compare the performance of mappings optimized with various cost functions under hot-spot traffic. For makespan measurement, almost each cost function becomes less predictable under hot-spot traffic. Except $f_5 \mid f_3$ and $f_7 \mid f_3$, mappings optimized with all other functions cease to improve after $n_T > 500$. $f_5 \mid f_3$ starts to show fluctuations after $n_T > 4000$. On the other hand, $f_7 \mid f_3$ still shows predictable improvements after $n_T > 5000$.

From the above results, we can conclude that mappings optimized with respect to $f_7 \mid f_3$ have the most predictable improvement under various traffic patterns. Thus, we will focus on evaluating this particular function.

Given the same P, the effect of increasing ΔT is shown in Figs. 5.8 and 5.9. Mappings are optimized with $f_7 \mid f_3$ after 5,000 trials. It is obvious that mappings optimized with $f_7 \mid f_3$ still improve over random mappings with significant margins for large ΔT's. It is found that even with $\Delta T = 250$, the margin of improvement is still more than 20% for both makespans and average latency measurements. Note that, for random mappings, the makespan decreases monotonically with increasing ΔT up to 180, showing that even when messages in P arrive in such a large time window, the network is still saturated. On the other hand, for mappings optimized with $f_7 \mid f_3$, the network becomes less congested when $\Delta T > 120$ and makespan tilts upward slightly with increasing ΔT's.

In Figs. 5.10 and 5.11, we again evaluate the performance of mappings optimized with $f_7 \mid f_3$, but assuming ℓ_i's given can be inaccurate to a certain degree. Here the optimization process still "sees" each message length as 20 flits long. But during simulations, for a given e, $0 \leq e \leq 1$, the actual length of m_i can be in $[20 - 20e,\ 20 + 20e]$. In Fig. 5.10, we can observe that when $e \leq 25\%$, the makespans of mappings are gradually improved with increasing n_T's. When $e = 50\%$ and $e = 75\%$, the performance becomes less predictable when $n_T < 3000$, but stabilized after $n_T > 3000$. For average latency measurement in Fig. 5.11, there are less fluctuations in all three curves. In any case, the mappings optimized with $f_7 \mid f_3$ still improve significantly over random mappings, i.e., $n_T = 0$.

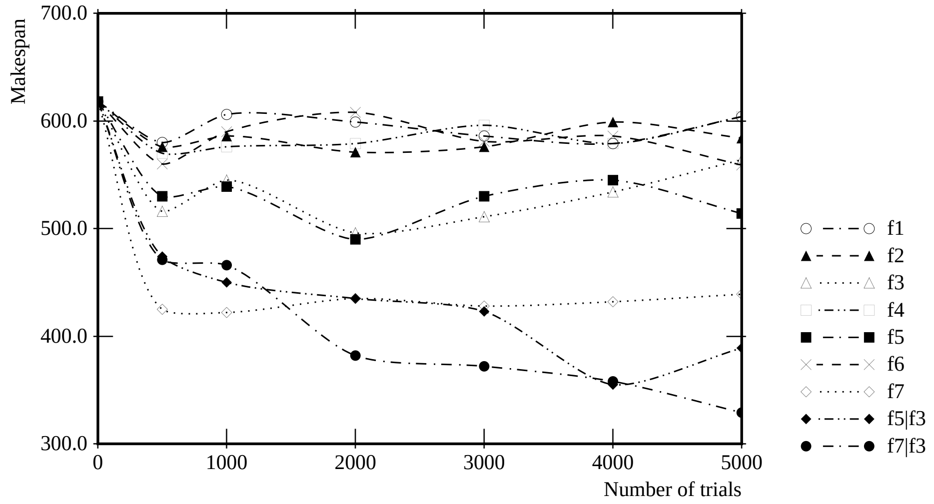


Figure 5.6: Makespan comparison of mappings optimized with various cost functions, hot-spot traffic.

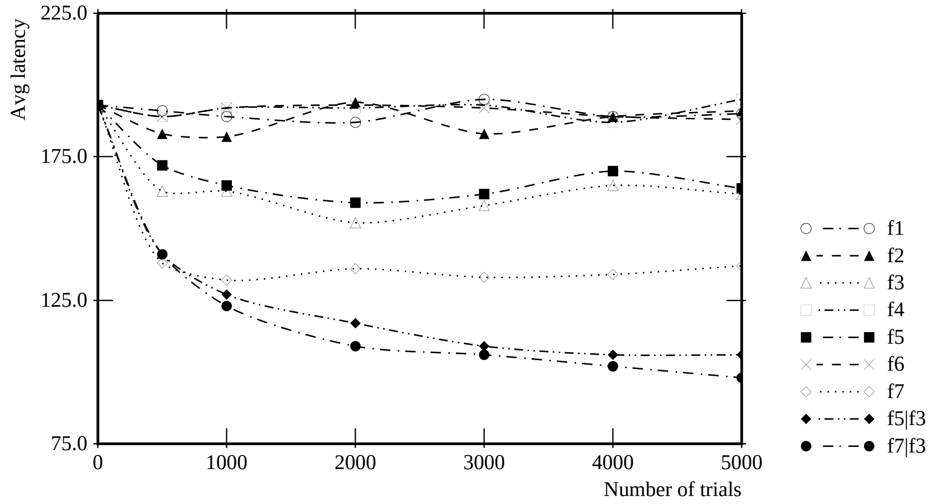


Figure 5.7: Average latency comparison of mappings optimized with various cost functions, hot-spot traffic.

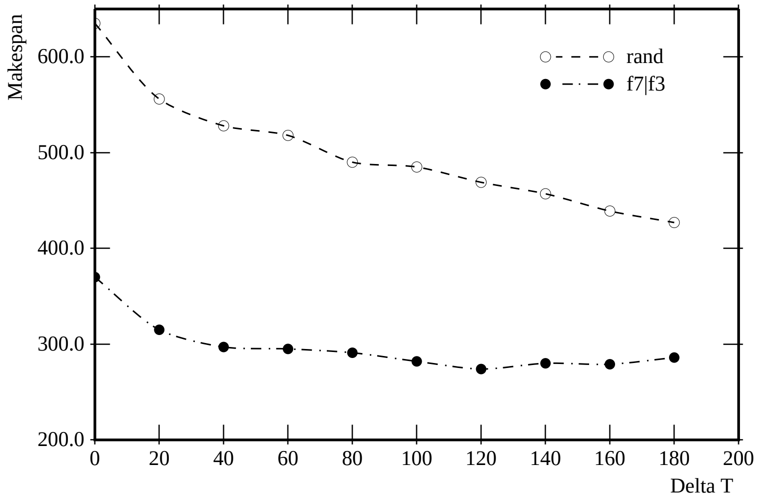


Figure 5.8: Makespan of mappings optimized with $f_7 \mid f_3$ versus varying ΔT.

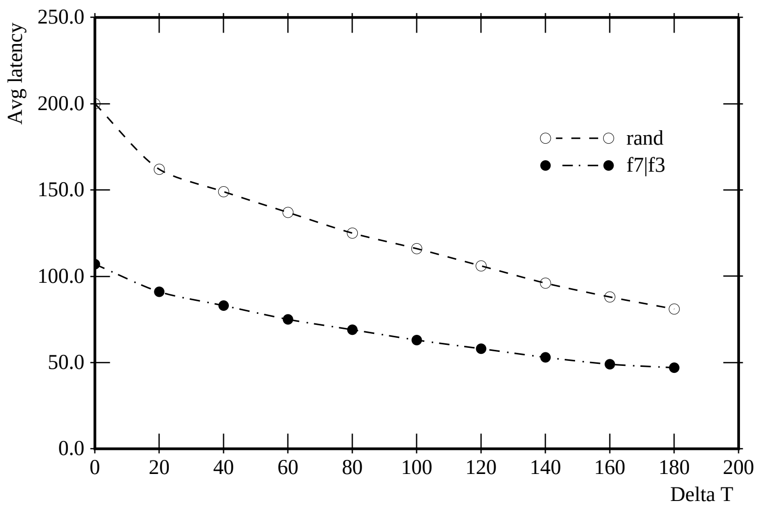


Figure 5.9: Average latency of mappings optimized with $f_7 \mid f_3$ versus varying ΔT.

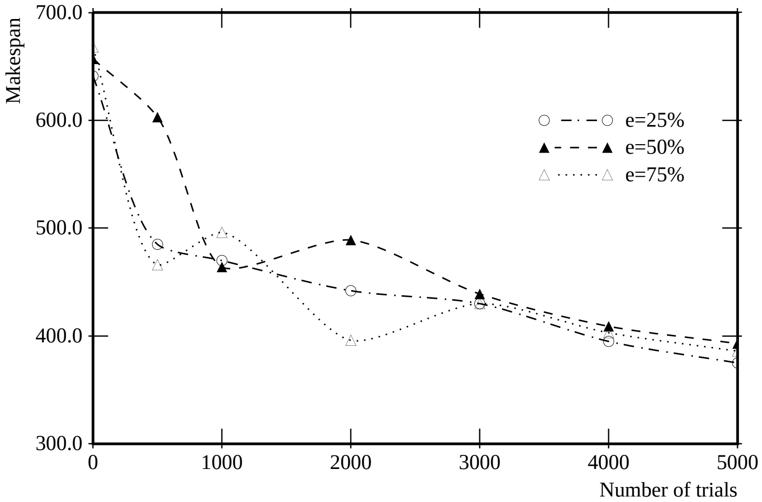


Figure 5.10: Makespan of mappings optimized with $f_7 \mid f_3$ using inaccurate ℓ_i's.

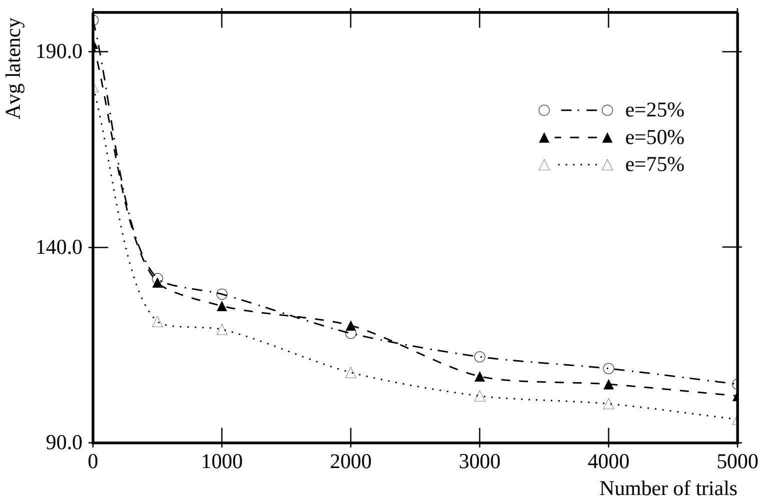


Figure 5.11: Average latency of mappings optimized with $f_7 \mid f_3$ using inaccurate ℓ_i's.

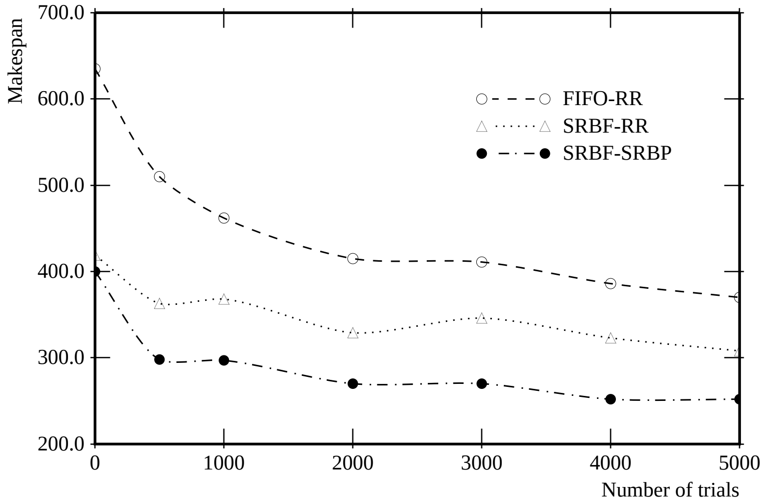


Figure 5.12: Makespan of mappings optimized with $f_7 \mid f_3$ under different flow-control strategies.

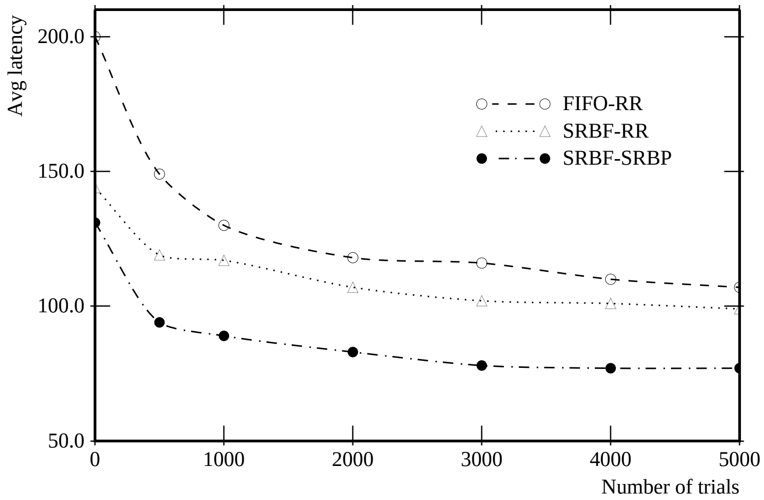


Figure 5.13: Average latency of mappings optimized with $f_7 \mid f_3$ under different flow-control strategies.

In Chapter 3, we have shown that by employing appropriate message-scheduling policies and flit-multiplexing methods, the performance of a virtual-channel network under concurrent communication traffic can be greatly improved. Here we will demonstrate that by applying these run-time flow-control mechanisms, the performance of mappings which are already optimized with $f_7 \mid f_3$, can be improved further. In Figs. 5.12 and 5.13, the performance measurements are shown for mappings optimized with $f_7 \mid f_3$ when executed on systems with various message-scheduling and flit-multiplexing combinations. "SRBF" denotes the message-scheduling policy which gives a higher priority to the message with the smallest remaining bandwidth. "SRBP" denotes the flit-multiplexing method giving a higher priority to the same type of messages as in SRBF. These two schemes are shown in Chapter 3 to perform particularly well. Also, to prevent deadlock, CTS lookahead is implemented with SRBP multiplexing.

These flow-control mechanisms can still improve the performance of mappings significantly. Though using SRBF scheduling alone can introduce some performance fluctuations when n_T is increased, it can still improve makespans by at least 12% and average latency by at least 10%. The combination of SRBF and SRBP can further improve the performance, especially the average latency. Also note that, when these flow-control mechanisms are used, the margin of improvement with increasing n_T's is narrowed. For example, mappings found with $n_T = 5000$ still outperform $n_T = 1000$, but when compared with the case using only FIFO-RR, the margin is greatly reduced. This shows that by using proper run-time flow controls, we may save some computing effort on finding optimized mappings.

In Figs. 5.14 and 5.15, we show the effect of applying the mapping optimization process and flow-control mechanisms on the performance of one set of communicating modules under uniform and hot-spot traffic, respectively. The mapping is optimized with $f_7 \mid f_3$ and $n_T = 5000$. It can be observed that given a mapping, different flow-control mechanisms will result in different rates of "energy"(remaining bandwidth) dissipation. Better flow-control not only results in a higher rate, but also a more linear behavior, and hence, a more predictable task communication response time. Furthermore, in the presence of hot-spot traffic, a good flit-multiplexing method like SRBP can reduce makespan dramatically by reducing the time the system spends in non-saturate regions, as shown in Fig. 5.15.

On the other hand, the mapping optimization process leads to lower "initial energy", and reduces the time needed to dissipate it. Note that it can work independently of the flow-control mechanisms, and their improvements on the performance can be additive. It

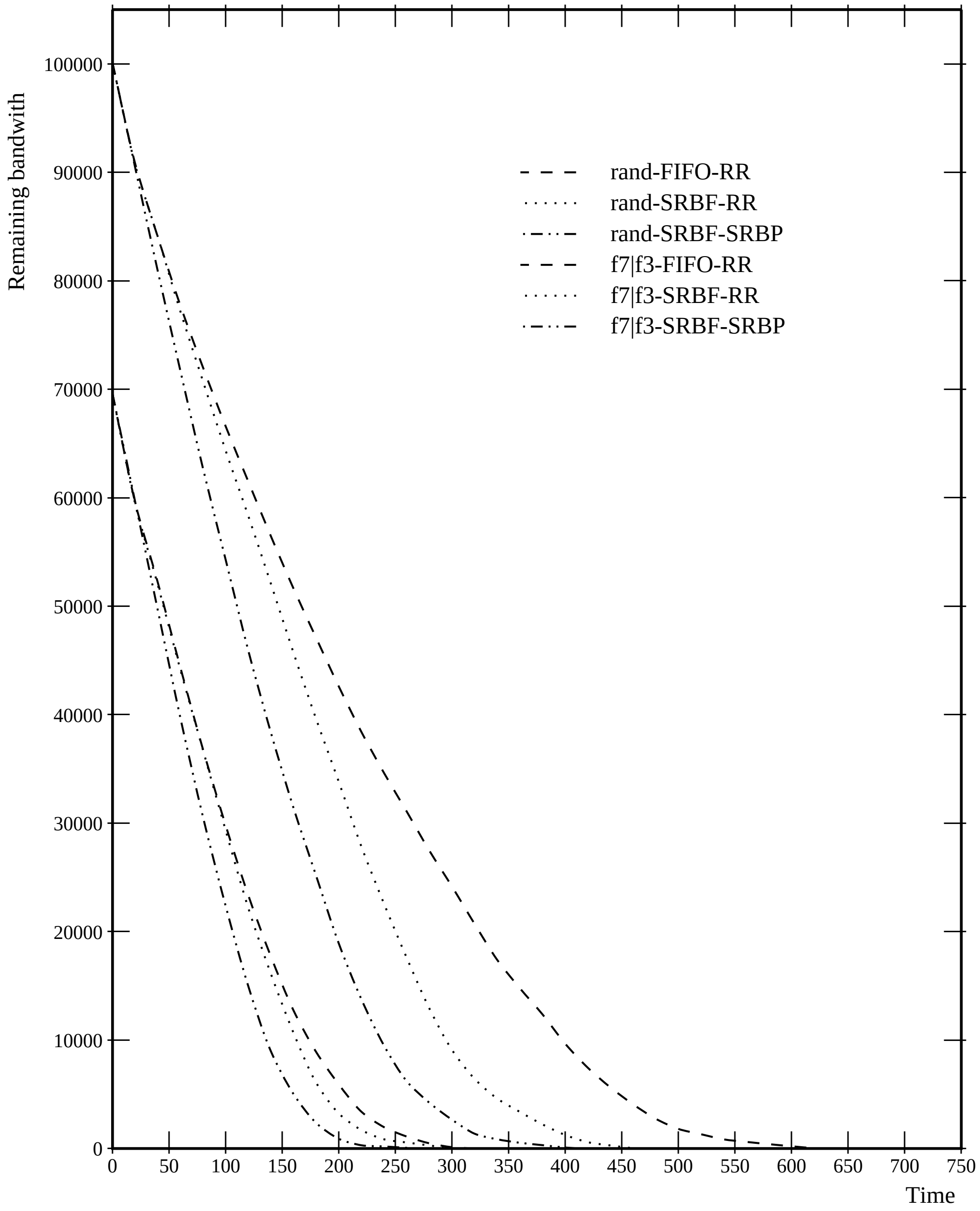


Figure 5.14: Plot of remaining bandwidth versus time, uniform traffic.

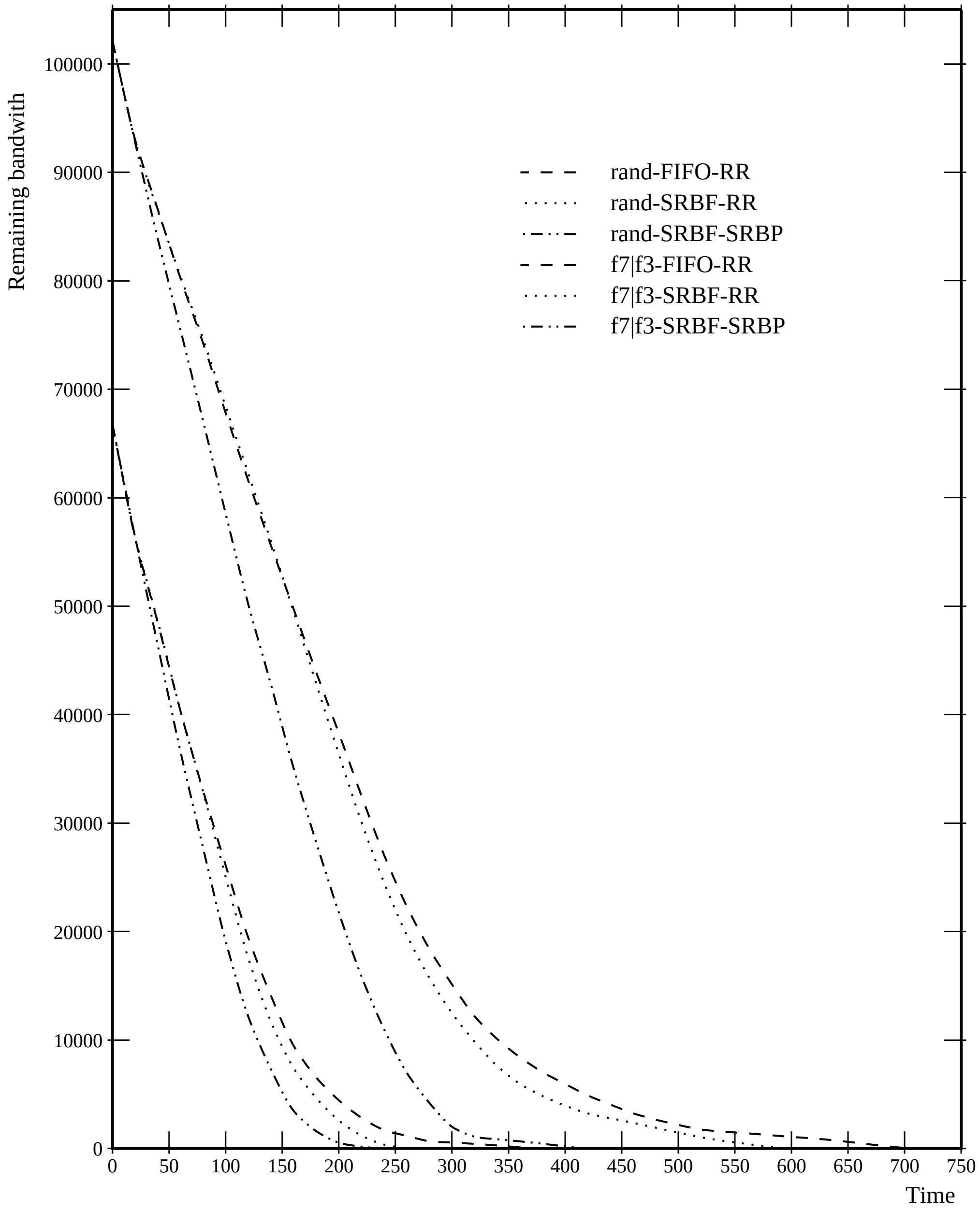


Figure 5.15: Plot of remaining bandwidth versus time, hot-spot traffic.

is also interesting to note that the amount of initial bandwidth is a good indication of the quality of mappings, especially when ΔT is small. In most of our inputs used here, when $\Delta T < 70$, a mapping with a smaller initial bandwidth almost always has a better makespan and average latency measurements. However, in most cases, given the same computing time, mapping optimized with f_3 actually has a higher initial bandwidth than $f_7|f_3$. The reason for this is that using f_3 alone, the simulated annealing process can be "trapped" in a local optimal much more quickly than using $f_7 \mid f_3$.

CHAPTER 6

MAPPING COMMUNICATING SUBCUBES IN A HYPERCUBE MULTICOMPUTER

6.1 Introduction

Subcube allocation — the problem of finding a subcube in a large target hypercube — has been studied extensively [17, 33, 3, 69, 97] under the assumption that incoming subcube requests are independent. The commonly-used objective of subcube allocation is to minimize hypercube fragmentation.

In certain applications, it may be necessary to cluster task modules into small groups, and each group is assigned to a subcube so as to minimize the distance of intra-group (or intra-subcube) communications. For example, in fault-tolerant applications where (application) task modules are replicated, the partial results obtained by replicas of a module can be sent to other modules only after they are voted on. Also, the execution of replicas needs to be synchronized for the synchronous voting of their results. (Asynchronous voting is known to be very hard due to the unpredictability of replica completion times [99].) Efficient communications among the replicas of each module are therefore crucial to the overall system performance.

There can still be inter-group (or inter-subcube) communications, which may become a major performance bottleneck if these communicating modules/subcubes are not carefully placed within the hypercube. For example, the embedding of TMR modules into the hypercube, as discussed in [70], requires each TMR to be embedded into a 2-dimensional subcube, Q_2. So, a task composed of communicating modules is embedded into a set of communicating Q_2's. If a pair of Q_2's communicate frequently with each other but are placed far apart, then a large amount of inter-subcube communication will result, which may in turn degrade intra-subcube communication performance, as both inter- and intra- subcube communications use the same network. We will in this chapter consider the problem of mapping communicating modules/subcubes in a hypercube by minimizing inter-subcube communication traffic.

The chapter is organized as follows. Section 6.2 introduces basic notation and assumptions, formally defines the objective function for subcube allocation, and states the optimization problem. In Section 6.3, we first discuss the special case of uniform subcube sizes, and derive some mathematical properties of the objective function. Methods are introduced to modify existing optimization algorithms to solve the problem. A special class of mappings, called *parallel mappings*, are found to be useful because of their unique properties. Also, it is proved that for some special cases, only parallel mappings need to be considered when one wants to find an optimal mapping. Thus, the optimization problem is greatly simplified. The general case of non-uniform subcube sizes is then discussed. Section 6.4 deals with sub-optimal mappings found with various heuristic algorithms. Through simulations, we show that when only sub-optimal mappings are considered, parallel mappings outperform non-parallel ones for most of the time.

6.2 Definitions, Assumptions, and Problem Statement

An n-dimensional hypercube, Q_n, consists of 2^n nodes which are connected in the form of a Boolean cube network. Each node is assigned a unique n-bit address, and two nodes are adjacent if and only if their addresses differ in exactly one bit position. We will henceforth use lower-case Greek letters to denote subcube addresses. Let Σ be a ternary symbol set $\{0, 1, *\}$, where $*$ represents don't care. Since each node in a Q_n is represented by n address bits, every subcube of the Q_n can be uniquely represented by a sequence of n ternary symbols in Σ, called the *address* of the corresponding subcube.

The Hamming distance between two subcubes $\alpha = a_0a_1...a_{n-1}$ and $\beta = b_0b_1...b_{n-1}$ of a Q_n is defined as

$$H(\alpha, \beta) = \sum_{i=0}^{n-1} h(a_i, b_i),$$

where $h(a_i, b_i) = 1$ if $(a_i, b_i \in \{0, 1\} \wedge a_i \neq b_i)$, and $h(a_i, b_i) = 0$ otherwise. For example, $H(00*, *11) = 1$, and $H(00*, 11*) = 2$.

We will assume the sizes of each communicating subcubes are known *a priori*, and communication events among those subcubes occur within a small *time window* [107, 109], i.e., messages are sent almost *concurrently*. A *weighted task graph* G will be used to represent the communication behavior among the subcubes within the time window. $G = (V, E)$, where V is the set of vertices each denoting a subcube, and $E = \{(v_i, v_j, w_{ij})\}$ the set of weighted, directed edges from v_i to v_j, where w_{ij} denotes the weight on the edge, and represents the length of the message from v_i to v_j.

We will first discuss a simple case where all subcubes are of the same dimension, using the subcube communication model defined for uniform-size subcubes as in [86, 18]. Given (v_i, v_j, w_{ij}), $w_{ij} > 0$, suppose $\phi_i = a_n a_{n-1} ... a_1$ is the subcube address which v_i is mapped to, and $\phi_j = b_n b_{n-1} ... b_1$ is the subcube address v_j is mapped to. We define an *instance* of subcube communication as each node in ϕ_i sends a message of length w_{ij} to another node in ϕ_j. For now, we are dealing only with uniform-sized subcubes of dimension d, so the number of messages sent is 2^d in each instance of communication. These messages are routed by the algorithm **Eq-subcube-route** proposed in [86], where a 1-to-1 mapping function is found between source and destination nodes, and the message between each source-destination pair is routed through a shortest path. Also, all messages in an instance of subcube communication are routed through edge-disjoint paths. From [18], the sum of lengths of these paths is given by $T(\phi_i, \phi_j) = M(\phi_i, \phi_j)2^d$, where $M(\phi_i, \phi_j)$ is defined as $M(\phi_i, \phi_j) = \sum_{i=1}^{n} m(a_i, b_i)$, where

$$m(a_i, b_i) = \begin{cases} 1 & \text{if } a_i = \bar{b}_i \neq * \\ 0 & \text{if } a_i = b_i \\ 1/2 & \text{otherwise.} \end{cases}$$

Therefore, we define the *bandwidth* of such an instance of subcube communication to be $w_{ij} T(\phi_i, \phi_j)$.

In Chapters 4 and 5, we have shown that for concurrently-communicating modules, the total bandwidth of a mapping is a good indicator of run-time performance. A mapping with a smaller total bandwidth almost always has better run-time performance, regardless of the underlying switching methods. We will henceforth use the *total bandwidth*, denoted by $\mathbf{\Phi}$, as our objective function. A formal definition is given below.

Given G and a target hypercube of dimension $n \geq \log(2^d \mid V \mid)$, i.e., it is large enough to accept all subcubes in V, our goal is to find a mapping of these subcubes into the target hypercube so that the total communication bandwidth of all communication instances is minimized. Formally, a mapping problem is described by a three tuple (G, d, n), and we want to minimize

$$\Phi = \sum_{v_i, v_j \in V} w_{ij} T(\phi_i, \phi_j),$$

where ϕ_i and ϕ_j are the addresses of subcubes v_i and v_j mapped to, respectively.

Before delving into details, we summarize the symbols used in this chapter as follows.

- d: the dimension of uniform-sized subcubes.
- D: the vector containing the dimensions of communicating subcubes when their sizes are non-uniform.
- $G = (V, E)$: the task graph.
- (G, d, n) or (G, D, n): a mapping problem under consideration.
- $H(\alpha, \beta)$: the Hamming distance between two addresses α and β.
- n: the dimension of the target hypercube.
- Q_n: a hypercube of dimension n.
- $T(\alpha, \beta)$: the total lengths of the paths taken by messages routed from subcube address α to β following the routing scheme in [86].
- v_i, v_j: vertices $\in V$ of G. Each v_i denotes a subcube.
- (v_i, v_j, w_{ij}): a weighted edge from v_i to v_j, which is an element $\in E$ of G.
- $\mid V \mid$: the number of elements in V.
- α, β: subcube addresses.
- $\mid \alpha \mid$: dimension of a subcube address α.
- ϕ_i, ϕ_j: the addresses of subcubes that v_i and v_j mapped to.
- Φ: the total bandwidth of a mapping.

6.3 Mathematical Properties

In this section we derive some mathematical properties which are important when finding optimal subcube mappings. We will first discuss the special case that all communicating subcubes are of dimension d.

6.3.1 Uniform-Size Subcubes

As in [18], we define the *frontier subcube* of α towards β, denoted by $\sigma_{\alpha\rightarrow\beta} = c_n c_{n-1} ... c_0$ such that $c_i = b_i$ if $a_i = * \wedge b_i \in \{0,1\}$, and $c_i = a_i$ otherwise. For example, if $\alpha = 00**$ and $\beta = 1*1*$ then $\sigma_{\alpha\rightarrow\beta} = 001*$. $\sigma_{\alpha\rightarrow\beta}$ contains all the nodes in α which are closest to β, i.e., the Hamming distance of each node in this subcube of α to β is exactly $H(\alpha, \beta)$.

Subcubes α and β are said to be *parallel* with each other, denoted as $\alpha \parallel \beta$, if $\mid \sigma_{\alpha\rightarrow\beta} \mid = d = \mid \alpha \mid = \mid \beta \mid$. Note that in the degenerate case, all Q_0's (individual nodes) are parallel with one another. It follows that if $\alpha \parallel \beta$, $T_{\alpha\beta} = 2^d H(\alpha, \beta)$. Under a *parallel mapping* all communicating subcubes are mapped to subcube addresses parallel with one another. In a parallel mapping, the expression for Φ can be rewritten as

$$\Phi = \sum_{v_i, v_j \in V} w_{ij} H(\phi_i, \phi_j) 2^d.$$

Therefore, if we only consider parallel mappings, minimizing the total bandwidth is equivalent to minimizing

$$\sum_{v_i, v_j \in V} w_{ij} H(\phi_i, \phi_j).$$

Note that if all subcubes are parallel, we can ignore the "don't cares", in subcube addresses when calculating their Hamming distances. The optimization problem for (G, d, n) is then reduced to finding optimal mappings for $(G, 0, n-d)$. This is just the optimization problem we treated in [107].

Parallel mappings also have the advantage that, even with the simplest (fixed-order) *e*-cube routing algorithm, all inter-subcube messages are routed through links that are never used for intra-subcube messages. For example, in Fig. 6.1(a), with *e*-cube routing, all inter-subcube messages are routed through links of the form $ab*$ or $*ab$, $a, b \in \{0,1\}$, but never through links $a*b$ which are used only for intra-subcube messages. But in the non-parallel mapping shown in Fig. 6.1(b), when both v_0's send messages to v_3's, fixed-order routing will either route a message through the link between the nodes which v_0's are mapped to, or the link between the nodes which v_3's are mapped to. This situation can only be avoided by introducing a more complex routing algorithm.

Note that for arbitrary, but not necessarily parallel α and β, given the frontier subcube $\sigma_{\alpha\rightarrow\beta}$, each of α and β can be partitioned into parallel subcubes of the same size as $\sigma_{\alpha\rightarrow\beta}$. So the computation of $T(\alpha, \beta)$ can be broken down into the evaluation of T's between parallel subcubes of dimension $\mid \sigma_{\alpha\rightarrow\beta} \mid$ within α and β. If $\mid \sigma_{\alpha\rightarrow\beta} \mid = 0$, then the evaluation of

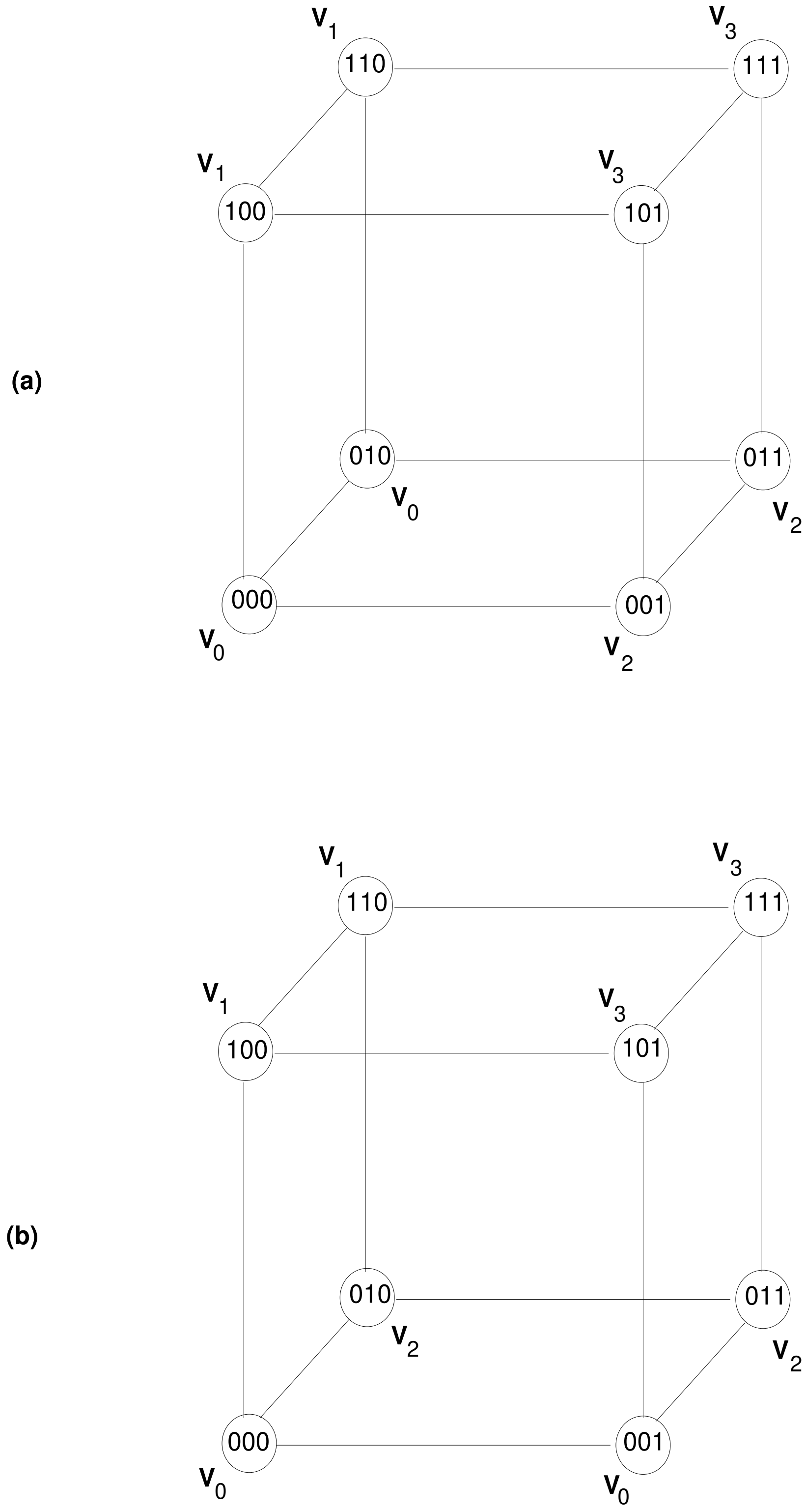


Figure 6.1: Two example mappings.

T degenerates into the case of evaluating the Hamming distances between many pairs of individual nodes in α and β. As an illustrative example, consider several Q_2's in a Q_4 as shown in Fig. 6.2. The two subcube addresses $01{*}{*}$ and $00{*}{*}$ are parallel, and $e^{01**,00**} = 2$. As for the two addresses $00{*}{*}$ and ${*}1{*}1$, $e^{00**,*1*1} = 1$, and $T(00{*}{*}, {*}1{*}1)$ can be expressed as $T(00*1, 01*1) + T(00*0, 11*1)$ or $T(00*1, 11*1) + T(00*0, 01*1)$. As for $00{*}{*}$ and ${*}{*}00$, $e^{00**,**00} = 0$, so $T(00{*}{*}, {*}{*}00)$ can only be expressed as in the definition of T, i.e., the sum of Hamming distances between individual nodes in these two subcubes.

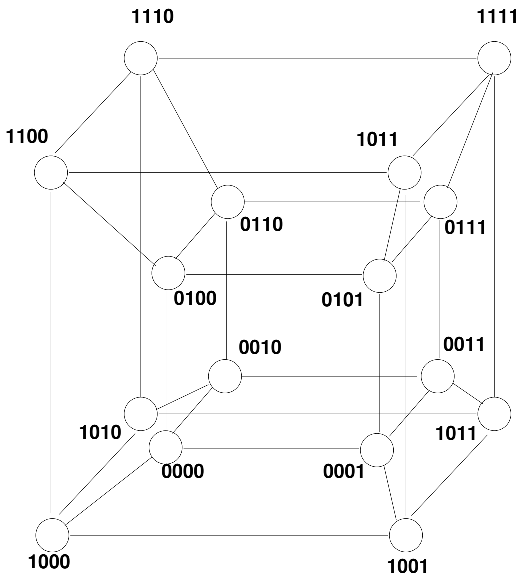


Figure 6.2: An example Q_4.

As a result, all mappings for the problem (G, d, n) can be expressed as parallel mappings for (G^{d-f}, f, n), $f \le f^*$, where G^{d-f} is some graph constructed from G (to be explained below), and f^* is the dimension of the *greatest common frontier*(GCF) subcube, which can be calculated by counting the number of common positions in which $*$'s appear in all subcube addresses. For example, we have a problem of $(G, 2, 4)$ with G given in Fig. 6.3, and we have a non-parallel mapping $v_0 \to 0{*}{*}0$, $v_1 \to {*}1{*}1$, $v_2 \to {*}0{*}1$ and $v_3 \to 1{*}{*}0$. Then this mapping can be expressed as parallel mappings of either 16 Q_0's or 8 Q_1's, which are smaller or equal to the size of the GCF subcube of the three Q_2's. G^{d-f} is just 2^{d-f} *disjoint* copies of G:

- $V^{d-f} = \{v_{i,x} \mid v_i \in V,\ 0 \le x < 2^{d-f}\}$
- $E^{d-f} = \{(v_{i,x}, v_{j,x}, w_{ij}) \mid (v_i, v_j, w_{ij}) \in E, 0 \le x < 2^{d-f}\}$.

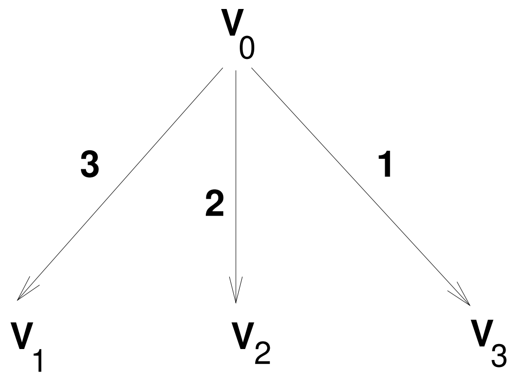


Figure 6.3: An example G.

Therefore, if the value of f is known and if an optimal parallel mapping for (G^{d-f}, f, n) is an optimal mapping for (G, d, n), then the optimization algorithm for $(G, 0, n)$ can be applied to solve (G, d, n). However, in general, the value of f in an optimal mapping of (G, d, n) is not known, so we have to consider the worst case of $f = 0$ and construct G^{d-0}, which will be denoted as G^d for simplicity.

Note that any mapping for (G, d, n) can be expressed as some mapping for $(G^d, 0, n)$, but the converse is not true. In other words, the set containing all mappings for (G, d, n) is a subset of the set containing all mappings for $(G^d, 0, n)$. Therefore, if we find an optimal mapping for $(G^d, 0, n)$, it could be useless since it may not be a *valid* mapping for (G, d, n), i.e., copies of certain v_i are not mapped into a Q_d. For example, let us consider $(G, 1, 3)$ with G given in Fig. 6.3 and its corresponding G^1 is just two identical G's. Fig. 6.4 shows an optimal mapping for $(G^1, 0, 3)$, but this mapping is not valid for $(G, 1, 3)$ since copies of v_0 are not mapped into a Q_1.

It is possible to avoid this problem by modifying G^d into G^d_X. We will add some extra edges to E^d, which are of the form $\{(v_{ix}, v_{i(x+1)}, X) \mid 0 \leq x < 2^d - 1\}$, and if $d > 1$, another edge $\{(v_{i(x+2^d-1)}, v_{ix}, X)\}$ is added. X is some sufficiently large number and its appropriate value is calculated by the method below.

When these edges are added, Φ of any mapping for $(G^d_X, 0, n)$ should be the value of Φ of some mapping for $(G^d, 0, n)$ plus M such that $M \geq \mid V \mid 2^d X$. The value X is chosen so that an optimal mapping found will always map all copies of v_i into a Q_d, $\forall v_i \in V$. And if $M \geq \mid V \mid 2^d X + X$, the mapping can never be optimal and is not valid for (G, d, n). So, we must have

$$\mid V \mid 2^d X + X + \Phi(opt(G^d, 0, n)) > \mid V \mid X 2^d + \Phi(wst(G, d, n)),$$

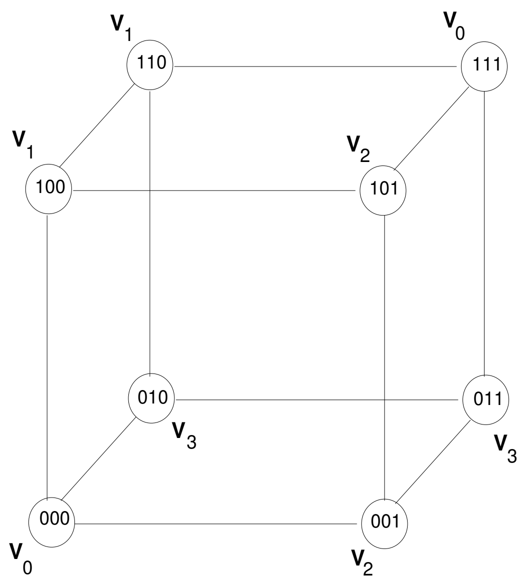


Figure 6.4: A mapping for $(G^1, 0, 3)$.

where $opt(G^d, 0, n)$ denotes the optimal mapping for $(G^d, 0, n)$ and $wst(G, d, n)$ the worst mapping for (G, d, n). So we must have $X > \Phi(wst(G, d, n)) - \Phi(opt(G^d, 0, n))$. We can substitute any upper-bound for $\Phi(wst(G, d, n))$ and any lower-bound for $\Phi(opt(G^d, 0, n))$ to obtain the value of X needed.

For example, to prevent an optimization algorithm from finding an invalid mapping as in Fig. 6.4, we construct G^1_X as in Fig. 6.5. An upper-bound of $\Phi(wst(G, 1, 3))$ is calculated by assuming the worst possible case that any pair of v_i and v_j are mapped 3 hops away from each other, which is the farthest distance in a Q_3. A lower-bound of $\Phi(opt(G^1, 0, 3))$ is calculated by assuming any pair of v_i and v_j are mapped adjacent to each other. So $X > 3 * 2(3 + 2 + 1) - 1 * 2(3 + 2 + 1) = 24$.

Therefore the optimization algorithm for $(G, 0, n)$ can find an optimal mapping for (G, d, n) by constructing G^d_X and apply the algorithm to $(G^d_X, 0, n)$. Since the optimization problem is NP-hard [107], the computation cost is much higher than the case of finding an optimal parallel mapping, where only the problem $(G, 0, n - d)$ needs to be considered. In what follows, we will show that for some special cases of G, an optimal parallel mapping is indeed optimal. Therefore, the optimization process can be greatly simplified.

We define a *sub-mapping* of a mapping for (G, d, n) as a set of node addresses in a Q_n which collectively contain a mapping for $(G, 0, n)$. Each mapping for (G, d, n) can be *partitioned* into 2^d such sub-mappings, and the node addresses which the copies of v_i mapped to must form a Q_d to make the mapping valid for (G, d, n). Note that we can

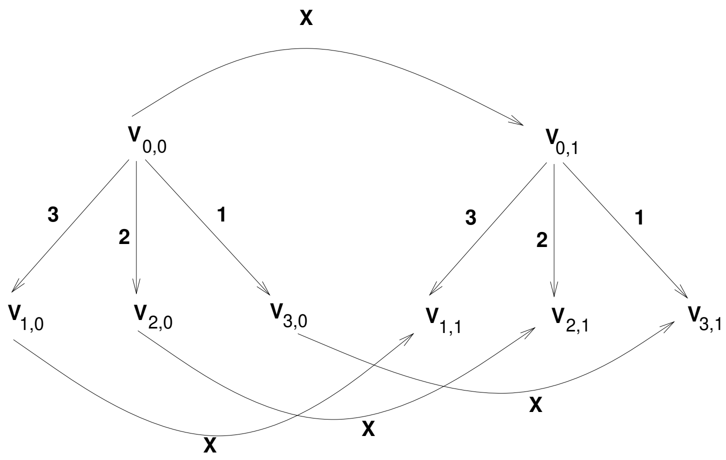


Figure 6.5: An example G_X^1.

partition a parallel mapping such that each of these 2^d sub-mappings lies within a Q_{n-d}, and each sub-mapping is a mapping for $(G, 0, n-d)$. Furthermore, an optimal parallel mapping for (G, d, n) can be partitioned into 2^d sub-mappings, each of which is an optimal mapping for $(G, 0, n-d)$. But this is not true in the case of non-parallel mappings. For example, in Fig. 6.6(a) we have an optimal parallel mapping for $(G, 1, 3)$ and in Fig. 6.6(b) a non-parallel mapping, with G again given in Fig. 6.3. A partition of each of the mappings is highlighted by the shaded nodes. In the parallel mapping, each sub-mapping lies within a Q_2 and is an optimal mapping for $(G, 0, 2)$. In the non-parallel mapping, sub-mappings are not mappings for $(G, 0, 2)$.

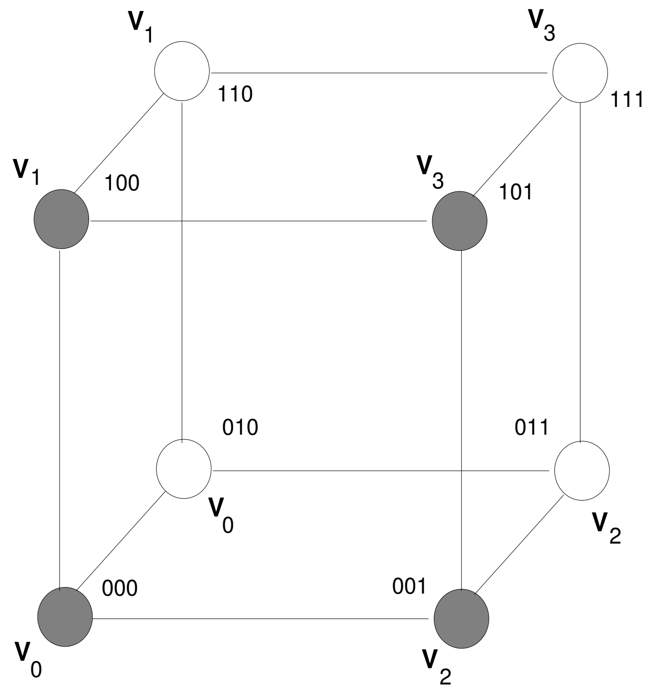
The following proposition is stated without giving the proof, which is trivial.

Proposition 1 *For a problem (G, d, n), if there exists a mapping better than an optimal parallel mapping, then the mapping can be partitioned into sub-mappings and there must be a sub-mapping whose Φ is smaller than that of $opt(G, 0, n-d)$, the optimal mapping for $(G, 0, n-d)$.*

We will show that, if G is a star-like graph, i.e., all edges in G directed into or out from just one vertex, then there is no such sub-mapping.

To facilitate the proof, let us first consider an example problem $(G, 1, 3)$ with G given in Fig. 6.3. The mapping in Fig. 6.6(a) is an optimal parallel mapping for this problem. To further improve the mapping, we need to have a mapping consisting of sub-mappings some of which have a smaller Φ than any sub-mapping of the optimal parallel mapping. Since each of the sub-mappings shown in Fig. 6.6(a) is optimal in $(G, 0, 2)$, a better sub-mapping

(a)



(b)

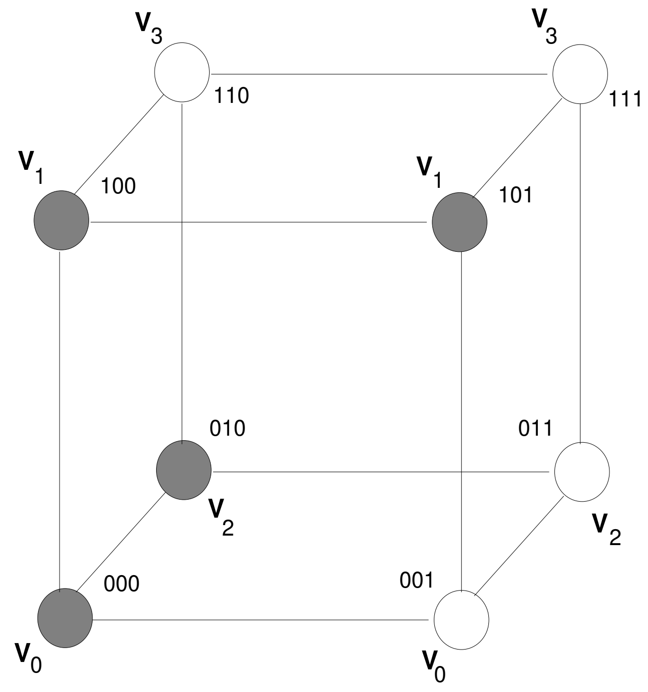


Figure 6.6: Two mappings for $(G, 1, 3)$.

must be some mapping in $(G, 0, 3)$. The only way to find a mapping for $(G, 0, 3)$ which has a smaller Φ than an optimal mapping for $(G, 0, 2)$ is to utilize the dimension not used in mappings for $(G, 0, 2)$. Hence we have in Fig. 6.7 an optimal mapping for $(G, 0, 3)$. This mapping is produced from re-mapping v_3 in node 101 which is 2 hops away from 000, to node 010 which is only 1 hop away from v_0, hence reducing Φ from 7 to 6. However, this forces v_0 in node 010 to be relocated to node 101. This leads to a mapping which is not valid for $(G, 1, 3)$, since the two nodes v_0 now mapped to cannot form a Q_1. On the other hand, if v_2 in node 001 is exchanged with v_0 in node 010, the resulting mapping shown in Fig. 6.6(b) is valid for $(G, 1, 3)$, but the sub-mapping does not have a smaller Φ. The same holds if we exchange v_1 in node 110 with v_0 in node 010. Therefore, the optimal parallel mapping in Fig. 6.6(a) is also a true optimal mapping for $(G, 1, 3)$.

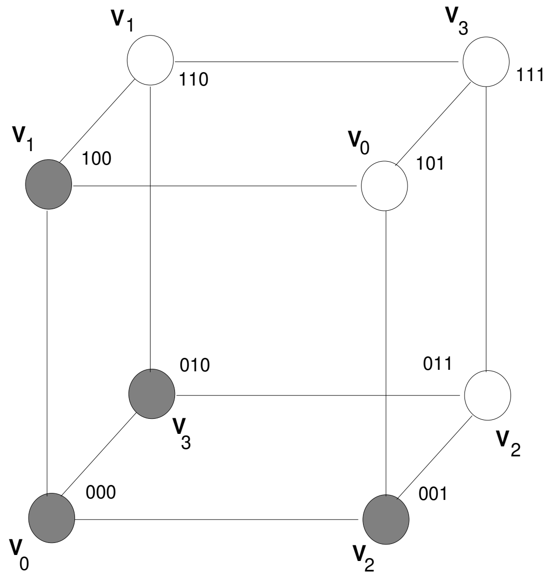


Figure 6.7: A sub-mapping which cannot be part of a valid mapping for $(G, 1, 3)$.

Lemma 1 *If G is a star, an optimal parallel mapping for a problem (G, d, n) must also be optimal among all mappings.*

Proof: An optimal parallel mapping can be considered as 2^d copies of the optimal mapping of the problem $(G, 0, n-d)$, each denoted as $opt(G, 0, n-d)$, and the 2^d nodes that a given v_i mapped to form a Q_d. The Φ value of this mapping can be expressed as $2^d \Phi^{opt}_{0,n-d}$, where $\Phi^{opt}_{0,n-d}$ is the Φ value of $opt(G, 0, n-d)$. Suppose there is a non-parallel mapping with a smaller Φ. This mapping can also be partitioned into 2^d copies of sub-mappings. There must exist at least one sub-mapping whose Φ is smaller than $\Phi^{opt}_{0,n-d}$. To satisfy this condition, this sub-mapping must be some mapping for $(G, 0, m)$, $n-d < m \leq n$.

Without loss of generality, let us assume v_0 to be the central vertex of the star in G and is always mapped to the address $0^{n-d}*^d$. In the optimal parallel mapping, consider the sub-mapping in $*^{n-d}0^d$ where v_0 is mapped to 0^n. Since G is a star, the value of Φ is determined only by the distance of v_i's to v_0. If there is a mapping for $(G, 0, m)$ with a smaller Φ than $opt(G, 0, n-d)$, then some v_i's in $opt(G, 0, n-d)$ must be re-mapped to the subcube $0^{n-d}*^d$. These addresses v_i's originally mapped to and 0^n must form a Q_d; otherwise the copies of v_0 cannot be re-mapped into a Q_d. However, if the original addresses which these v_i's mapped to can form a Q_d, then re-mapping these v_i's into $0^{n-d}*^d$ cannot lower Φ. Otherwise, the original sub-mapping cannot be optimal in $(G, 0, n-d)$. A contradiction. □

We are still unable to prove the general case of arbitrary task graphs. However, in enumerations for many low-dimensional cases, we could not find better non-parallel mappings than optimal parallel mappings. It is our conjecture that optimal parallel mappings are indeed optimal.

6.3.2 Non-uniform Size Subcubes

In Section 6.3.1, we have discussed the mapping problem based on the assumption that all communicating subcubes are of the same size. Here we will consider the general case where subcube sizes need not be uniform.

For (v_i, v_j, w_{ij}), $w_{ij} > 0$, suppose α is the address to which v_i is mapped, and β is the address v_j is mapped to. We generalize the definition of an instance of subcube communication as follows.

- If $\mid \alpha \mid < \mid \beta \mid$, then an instance of communication from α to β consists of instances such that α communicates with each distinct $Q_{|\alpha|}$ contained in β.

- If $\mid \alpha \mid > \mid \beta \mid$, then an instance of communication from α to β consists of instances such that each distinct $Q_{|\beta|}$ in α communicates with β.

For example, in a Q_4, an instance of communication from $\alpha = 00{*}0$ to $\beta = 0{*}{*}1$ consists of an instance of communication from $00*0$ to $00*1$, *and* an instance from $00*0$ to $01*1$. Or if we partition $0**1$ in another way, it consists of instances from $00*0$ to $0*01$ and to $0*11$. On the other hand, an instance from β to α consists of similar instances as above while messages are sent in opposite directions. For arbitrary α and β, $T(\alpha, \beta) = T(\beta, \alpha)$

$= M(\alpha,\beta)2^{\max(|\alpha|,|\beta|)}$, where $M(\alpha,\beta)$ is defined as before in Section 6.2. One can observe that how the larger subcube is partitioned does not affect the value of $T(\alpha,\beta)$ or $T(\beta,\alpha)$.

Now, we can describe the subcube mapping problem with a three-tuple (G,D,n), where $D=[d_0,d_1,...,d_{|V|-1}]$ is a vector specifying the dimensions of subcubes for $v_0,v_1,....,v_{|V|-1}\in V$.

Here we define $\alpha \parallel \beta$ if $|\ \sigma_{\alpha\rightarrow\beta}\ |=\min(|\ \alpha\ |,|\ \beta\ |)$, i.e., the frontier subcube is just the smaller of the two. This is consistent with the uniform-size case, where two addresses are parallel if the frontier subcube is either of the subcubes. In the degenerate case, an individual node address is parallel to any other subcube addresses. A parallel mapping is the one that all subcubes are mapped to addresses parallel to one another. So if f^* is the dimension of the GCF subcube in a parallel mapping, $f^*=d_{min}$, i.e., the smallest $d_i\in D$. While in a non-parallel mapping, $f^*<d_{min}$.

One important property of parallel mappings in the uniform-size case is that to find an optimal parallel mapping for (G,d,n), we only need to find an optimal mapping for $(G,0,n-d)$. For the case of non-uniform subcube sizes, we need to make some modification.

A parallel mapping for (G,D,n) can be expressed as a parallel mapping for (G',d_{min},n), where G' is constructed from G by partitioning each subcube into one or more $Q_{d_{min}}$'s. Formally, we have $V'=\{v_{ix}\mid v_i\in V, 0\le x<2^{d_i-d_{min}}\}$. For each $(v_i,v_j)\in E$, if $d_j>d_i$, then

$$\{(v_{ix},v_{jx},w_{ij}),(v_{ix},v_{j(x+1)},w_{ij}),...,(v_{ix},v_{j(x+k-1)},w_{ij})\mid 0\le x<2^{d_i-f}, k=2^{d_j-d_i}\}\subset E',$$

else if $d_j<d_i$, then

$$\{(v_{ix},v_{jx},w_{ij}),(v_{i(x+1)},v_{jx},w_{ij}),...,(v_{i(x+k-1)},v_{jx},w_{ij})\mid 0\le x<2^{d_j-d_{mini}}, k=2^{d_i-d_j}\}\subset E'.$$

For example, suppose G is given as in Fig. 6.3, a mapping for $(G,[1,2,2,1],4)$ is shown in Fig. 6.8, which can be expressed as a mapping for $(G',1,4)$ with G' in Fig. 6.9. However, the set of parallel mappings for (G',d_{min},n) may contain some mappings which are not valid for (G,D,n). For example, Figs. 6.10(a) and (b) are both valid mappings for $(G',1,4)$, but only the one in Fig. 6.10(a) is valid for $(G,[1,2,2,1],4)$. So if we apply the optimization algorithm to find an optimal parallel mapping for (G',d_{min},n), which is equivalent to finding an optimal mapping for $(G',0,n-d_{min})$, the mapping may not be valid for (G,D,n). The remedy for this is to adopt the method in Section 6.3.1 to modify the task graph. By adding extra edges in G' with sufficiently large weights, we can force copies of any given v_i to be mapped into a Q_{d_i}.

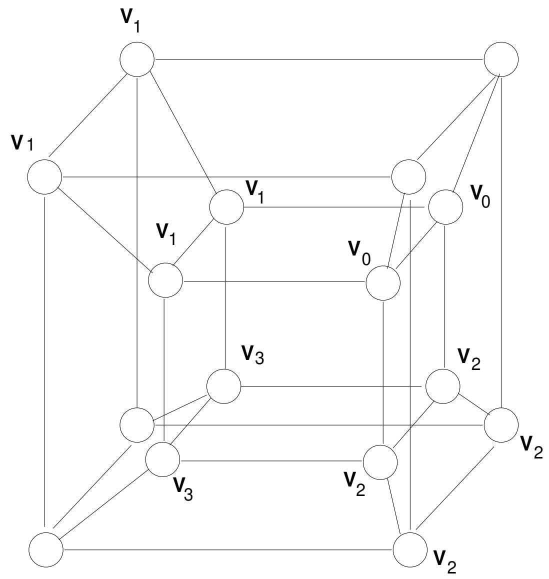


Figure 6.8: A mapping for $(G, [1, 2, 2, 1], 4)$.

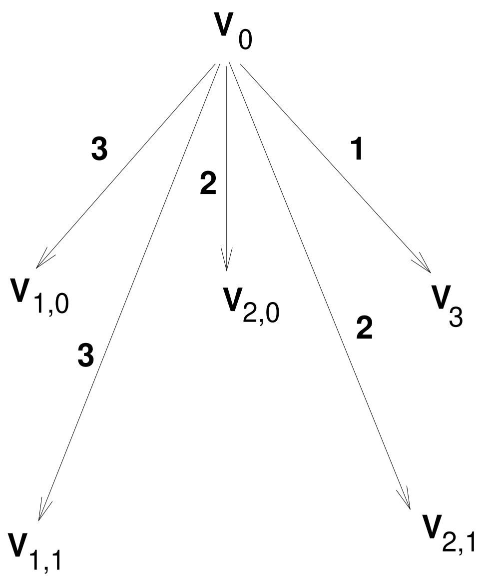
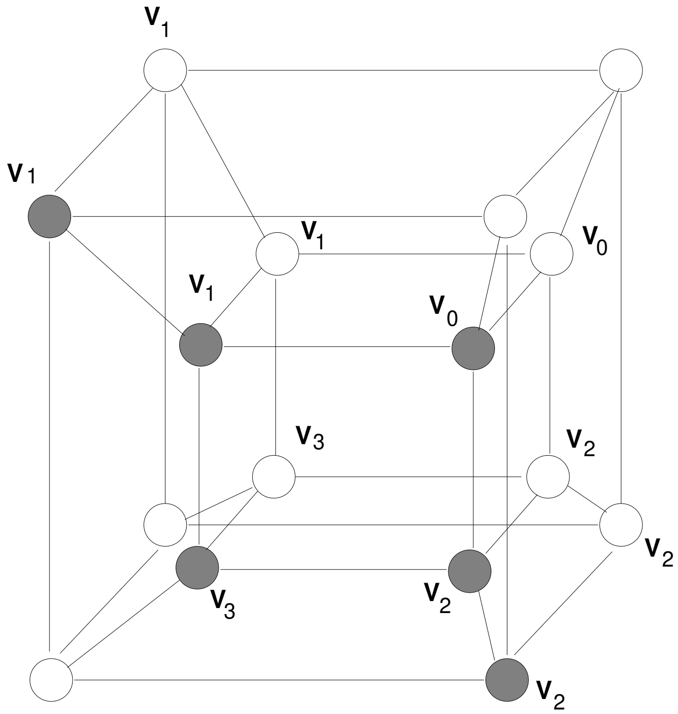


Figure 6.9: A G' constructed from G.

(a)



(b)

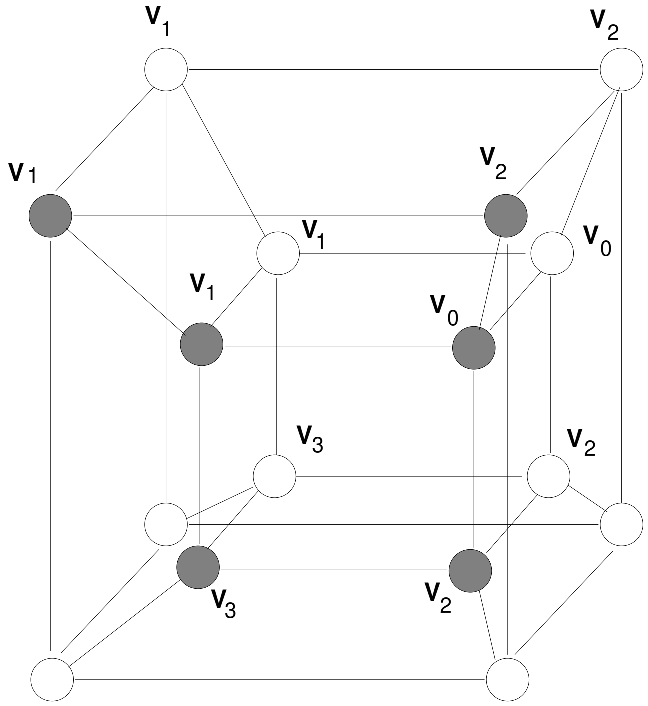


Figure 6.10: Example mappings for non-uniform subcube sizes.

For finding an optimal mapping, not necessarily a parallel one, a similar method can be used. A problem (G, D, n) is treated as $(G'^{d_{min}}_X, 0, n)$, where $G'^{d_{min}}_X$ is constructed from G' defined above by following the modifications mentioned in Section 6.3.1.

Similar to Lemma 1, we have the following lemma for certain special cases of the non-uniform size problem.

Lemma 2 *If G is a star where v_0 is the central vertex, and $d_0 = d_{min}$, then an optimal parallel mapping for a problem (G, D, n) must also be optimal among all mappings.*

Proof: Suppose we have an optimal parallel mapping for (G, D, n). It can be partitioned into $2^{d_{min}}$ sub-mappings each of which is a mapping for $(G', 0, n - d_{min})$, where G' is constructed from the above rules by partitioning subcubes into $Q_{d_{min}}$'s. Since $d_0 = d_{min}$, G' is still a star. The proof follows the same argument as in the proof of Lemma 1. □

6.4 Heuristic Mapping Strategies

In Section 6.3, we have discussed the mathematical properties of subcube mappings, and discussed a strategy to find an optimal mapping using a modified version of existing optimization algorithms developed for a simpler mapping problem. However, the complexity of these optimization algorithms remains exponential, and hence for large problem sizes, we need some heuristic algorithms to find good sub-optimal mappings.

We have shown that an optimal parallel mapping is also optimal among all mappings for certain special cases, and it is our conjecture that optimal mappings is indeed optimal. However, this does not imply that *all* parallel mappings are better than non-parallel mappings. Therefore, two important questions arise: When we use a heuristic algorithm to find a good sub-optimal parallel mapping, will this sub-optimal parallel mapping be worse than most non-parallel mappings? Do we need to consider all possible mappings, and not just parallel mappings when looking for a good sub-optimal mapping?

In this section, we investigate several heuristic methods, and compare the performance of parallel and non-parallel mappings found with each heuristic. We also confirm that optimization of mappings with respect to Φ improves several other performance parameters as well. We will focus on the discussion of the case of uniform-size communicating subcubes, since we have shown that mapping variable-size subcubes can always be reduced to a uniform-size subcube mapping problem.

6.4.1 Fixed-Size Target Hypercubes

We first study the case where the target hypercube dimension is fixed at n.

The simulated annealing method [72] is shown to be an effective algorithm for finding near-optimal solutions to NP-hard task-mapping problems [35, 108]. In [108], we investigated a simulated annealing method optimization process for finding good sub-optimal mappings for the problem $(G, 0, n)$. The implementation of the simulated annealing method here is based on parameters selected with a similar criterion as in [35]. We set the *initial temperature* $T_0 = 30$, the *new temperature* $T_{new} = 0.95T$, where T is the temperature in the last iteration. The *freezing point* is set so that a move increasing the objective function by a unit value has an acceptable probability of 2^{-31}. The perturb function is given by performing random 2-opt exchanges [2] on the original mapping. Since each instance of 2-opt exchange takes approximately the same amount of computing time, the expected computing time of an optimization process can be normalized and expressed as the average number of exchanges performed. Given the above parameters, the optimization process is found to terminate after 3000 ± 500 exchanges on 90% of inputs used in producing the data presented here.

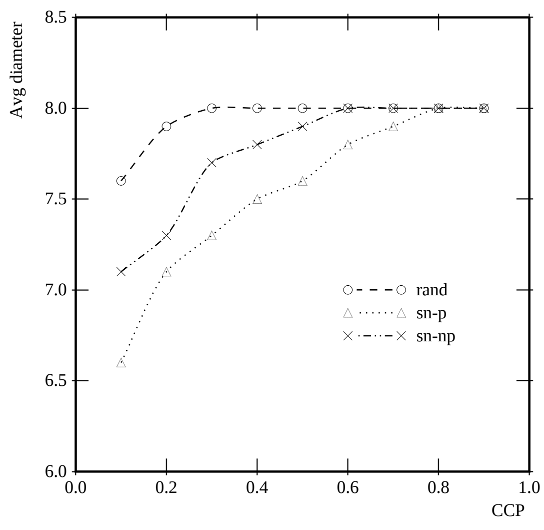


Figure 6.11: Average diameter versus CCP.

In Table 6.1, we compare the performance of parallel mappings (sn-p) and non-parallel mappings (sn-np) found with the simulated annealing method. For sn-p mappings, the initial mapping is a random parallel mapping, and only exchanges among parallel subcubes are allowed. For sn-np mappings, only non-parallel mappings are considered. The inputs are

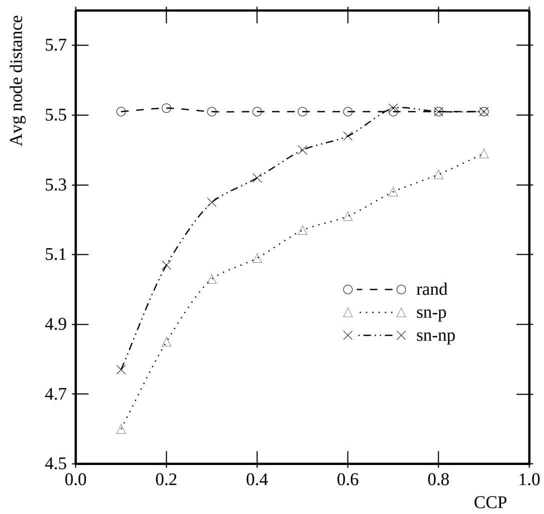


Figure 6.12: Average node distance versus CCP.

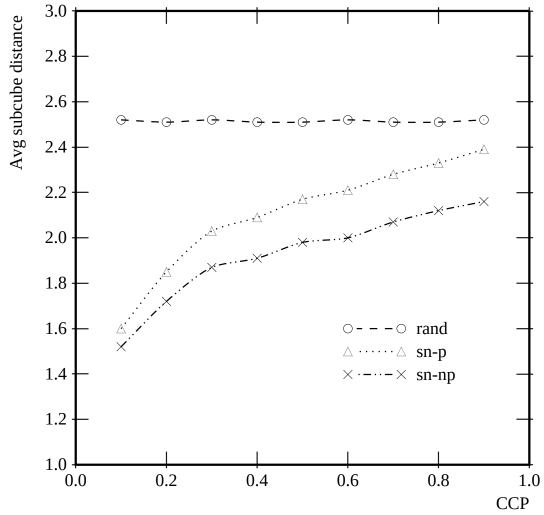


Figure 6.13: Average subcube distance versus CCP.

generated with $\mid V \mid = 25$, $prob(w_{ij} > 0) =$ CCP (Concurrent Communication Probability), i.e., the probability that v_i communicates with v_j in the time window considered. w_{ij} is set to 20 when $w_{ij} > 0$, and each subcube is of dimension $d = 3$. Each data point is obtained by averaging results from 10,000 iterations. Deviation from the mean values is found to be reasonably small ($< 3\%$).

CCP	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
rand	13472	25952	37888	49280	61056	72640	82368	96752
sn-p	8624	19072	30608	41216	52880	63968	76224	89344
sn-np	9552	21344	34016	45536	58464	70480	81432	95856

Table 6.1: Φ of mappings found by various strategies.

It is obvious that both sn-p and sn-np mappings improve over random mappings significantly when CCP is small. sn-p mappings have more than 15% improvement over random mappings when CCP < 0.6. Also, sn-p mappings outperform sn-np mappings consistently ($> 10\%$) for all CCP values. This is even more prominent when CCP > 0.5, where the margin between sn-np and random mappings narrows down.

When w_{ij}'s are constant as in our simulations, the optimization of Φ can also improve several important performance parameters of a mapping. The *communication diameter* of a mapping is the largest Hamming distance between two nodes belonging to a pair of communicating subcubes. The *average communication distance* is the average Hamming distance of all pair of nodes involved in inter-subcube communications. The *average subcube distance* is the average Hamming distance of all communicating subcubes. In Figs. 6.11 to 6.13, we show the three performance parameters plotted against CCP for the mappings found. Obviously, sn-p mappings outperform random and sn-np mappings in all cases, except in the case of average subcube distance, where sn-np has a lower value for various CCP values. This shows that non-parallel mappings are better only in minimizing the Hamming distance between subcubes, not necessarily between nodes.

6.4.2 Strategies for Unconstrained-Size Target Hypercubes

We now assume the target hypercube size to be unconstrained for mapping subcubes.

As shown in Section 6.3, $\Phi(opt(G, d, n + 1)) \leq \Phi(opt(G, d, n))$, so given some task graphs, one may obtain better mappings for a larger target hypercube. Since processor nodes are often abundant in modern distributed-memory systems, the number of nodes used for embedding a task should not be a major limiting factor. A mapping which needs a larger target hypercube but demonstrates better communication performance may actually be more attractive. Determining the size of the smallest target hypercube needed for an optimal mapping is also a hard problem even when G's are restricted to trees, as shown in [114]. We will investigate a fast heuristic which finds a spanning tree of each connected component of G and embed the tree into a target hypercube. The basic idea is to try to find a *maximal* spanning tree such that the sum of weights on the tree edges is maximized, and then the tree is isomorphically embedded into a target hypercube of appropriate size, so that these heavily-communicating subcubes are mapped adjacent to another. Finding the *minimal* spanning tree of G is easily solved in polynomial time [13], but finding a maximal one is itself NP-hard.

In the "df" heuristic, G is visited in a depth-first manner. On a vertex v_i, the edge leading to an unvisited vertex v_j with the largest value of $w_{ij} + w_{ji}$ is added to the tree. In df-p, neighboring nodes in the tree is mapped only to parallel subcubes of Hamming distance one to each other, while non-parallel subcubes of Hamming distance one to each other are allowed in df-np. Intuitively, allocating two non-parallel subcubes is easier than locating two parallel ones. So a possible advantage of mapping non-parallel subcubes is to reduce the size of the target hypercube needed for a mapping, and possibly smaller Φ's. In the "bf" heuristic, G is visited in a breadth-first manner. A vertex is first picked so that the sum of weights on all outgoing and incoming edges is the largest. The process is then repeated for each of the unvisited vertices connected to this vertex, and so on, until all vertices in G are covered.

In Table 6.2, we compare the performance of mappings found by different heuristics. The inputs used are the same as those used in obtaining the data in Table 6.1. The minimal target hypercube dimension required is 8. Table 6.3 shows the average dimension of the target hypercube required for the corresponding mappings in Table 6.2.

In both bf and df, the non-parallel mappings all result in worse performance, although the target hypercube sizes are reduced slightly over parallel mappings. bf-p mappings have about the same performance as df-p when CCP < 0.5, and only slightly improve over df-p for higher CCP values. However, the bf approaches result in a large increase of required target hypercube size. It is interesting to note that the df approach actually needs a smaller target hypercube as CCP increases. This is because as CCP gets larger, G has a higher probability to become Hamiltonian, and its spanning tree becomes "narrower" with lower vertex degrees. For the bf approaches, the trend is reversed and larger target hypercubes are needed for larger CCP values. Note that no matter which heuristic is used, the actual number of nodes occupied is still the same. The difference is in the size of the target hypercube that becomes fragmented. A heuristic requiring a larger target hypercube will lead to fragmentation of a larger hypercube.

CCP	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
df-p	8240	21344	34240	46272	58320	70400	82992	95584
df-np	8292	21731	34923	47073	59820	72183	84143	98829
bf-p	8112	21456	35440	46704	57808	68096	79184	91056
bf-np	8128	21497	35513	46915	58039	69011	81121	93746

Table 6.2: Φ of mappings found by different heuristics.

CCP	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
df-p	9.1	8.5	8.2	8.1	8.0	8.0	8.0	8.0
df-np	8.9	8.3	8.1	8.0	8.0	8.0	8.0	8.0
bf-p	9.1	10.0	11.3	12.7	13.8	15.0	15.0	15.0
bf-np	9.0	9.8	10.4	11.1	11.9	12.8	13.5	14.2

Table 6.3: Average size of required target hypercubes for various heuristics.

CHAPTER 7

CONCLUSIONS AND FUTURE DIRECTIONS

7.1 Summary

In this dissertation, we have addressed several important issues in improving the performance of multicomputer networks in the presence of concurrent communication traffic. The proposed strategies have been shown to significantly improve communication performance during task execution. In Chapters 2 and 3, we focused on flow-control mechanisms to improve communication performance during the task execution. On the other hand, in Chapters 4 to 6, we addressed the issues of mapping task modules onto the target network before executing the task.

Chapter 2 dealt with optimizing the performance of interprocessor-communication in a hypercube multicomputer equipped with SPIDER-like adapters under concurrent traffic. Branch-and-bound algorithms were developed to find optimal schedules for various switching methods under the non-adaptive e-cube routing algorithm. Though computationally expensive, these optimal schedules serve to measure the effectiveness of various scheduling policies. A centralized path selection algorithm based on the simulated annealing method was also developed and serves as a reference for evaluating distributed routing algorithms. Several distributed message scheduling policies under the e-cube routing algorithm were examined for systems with message switching, circuit switching, and virtual cut-through. In our simulations, the Largest Remaining Bandwidth First (LRBF) scheduling policy was found to be very effective. It could approach the performance of optimal schedules in many situations. Being a distributed scheduling policy, it was also more practical and computationally much less expensive than centralized approaches.

A low-complexity adaptive routing algorithm, called the Progressive Adaptive (PA) algorithm, was also evaluated. When combined with the LRBF scheduling policy, this routing algorithm was found to be very effective in improving performance over the e-cube algorithm. It outperformed the more complex DFS routing under heavy traffic conditions, and could closely match the performance of centralized, near-optimal approaches.

Though we evaluated network performance under transient communication loads, it was shown that an improvement in transient performance almost always offered better steady-state performance.

In Chapter 3, we evaluated the performance of several message-scheduling policies and flit-multiplexing methods in a mesh network with wormhole switching and virtual channels. We focused on w-meshes (k-ary 2-cubes) and f-meshes ($k \times k$ meshes) with e-cube routing subject to concurrent traffic. We mainly dealt with low-complexity flow control mechanisms. Simulations were performed for three message-scheduling policies, FIFO, SRBF and LRBF, and their combinations with flit-multiplexing methods such as demand-driven (DD) allocation, CTS lookahead, and priority-based multiplexing.

We used two performance measures in evaluating these configurations: $\hat{t}$, the makespan of a communication mission, and $\bar{t}$, the mean latency. It was found that DD allocation and CTS lookahead are both essential to minimize the waste of physical bandwidth. With a small amount of extra hardware, SRBF message scheduling and the SRBP flit multiplexing can improve network performance significantly. Also, w-meshes, though with more communication resources, may perform worse than f-meshes in certain situations.

Using a simple objective function, we formulated and solved the problem of mapping a task which is composed of multiple interacting modules into a binary hypercube in Chapter 4. The goal was to optimize task communication performance, measured in communication makespan. Due to the difficulties in optimizing this objective directly, a function called communication bandwidth is proposed. By minimizing this function, we could find assignments with the optimal communication performance using heuristic combinatorial techniques. Several heuristics that find mappings by minimizing communication bandwidth are implemented and comparatively evaluated. The mappings found with these algorithms are also evaluated with simulations to assess their performance during task execution. It has been shown that for communication-bound tasks, they make significant improvements over random assignments with respect to an actual communication performance measure, i.e., the communication makespan. We also analyzed the case where an alternative routing algorithm like DFS routing is used. Our task mapping strategy is again shown to work well in this case.

In Chapter 5, we have addressed the problem of mapping concurrently concurrently-communicating modules into a mesh multicomputer with wormhole switching and virtual channels. Our objective is to optimize the makespan and average latency of these messages exchange among modules. It has been shown that a direct optimization to the performance objective itself is not practical. We investigated several simplified cost functions for the simulated annealing method. The effectiveness of these proposed cost functions are compared by using a flit-level simulation program to access the actual run-time performance of the mappings optimized with each cost function when approximately the same amount of computing time is given. The cost function $f_7 \mid f_3$, has been found to be quite effective. Mappings optimized with it have been shown to be consistently outperform the others. Also performance of mappings can be continually improved with the increase in computing time. We also showed that the run-time performance of optimized mappings can be further improved when on-line flow-control mechanisms are implemented.

We addressed the problem of mapping a set of communicating subcubes in a hypercube by minimizing inter-subcube communication traffic in Chapter 6. The communication model we used was based on the one proposed in [86, 18] for routing messages between subcubes of the same size. Our objective was to minimize the total inter-subcube communication bandwidth.

We first considered the case where all subcube sizes were identical. Several important mathematical properties of this type of mappings were derived. Methods were proposed to modify existing algorithms to find an optimal mapping. Parallel mappings were found to have certain desirable properties, and required less computation cost to find. It was also shown that for some special cases optimal parallel mappings were indeed optimal among all mappings. We then generalized our discussion to include communications among non-uniform size subcubes.

We later showed by simulations that in heuristic algorithms such as simulated annealing methods and other fast heuristics, parallel mappings still outperformed non-parallel mappings in most cases. Also, in our simulations, optimizing the proposed objective function also lead to improvements in several other performance parameters.

7.2 Contributions

The main contributions of this dissertation are summarized as follows.

- We proposed practical on-line and off-line strategies to reduce traffic congestion under concurrent traffic arrivals. Most of the proposed strategies can be readily implemented using simple hardware circuits or low-complexity software code on existing multicomputers. In binary hypercubes equipped with large-buffer switching methods, we have studied the effects of combining routing algorithms with message-scheduling policies. In the past, researchers had mostly focused on complex routing algorithms without considering the importance of on-line message scheduling. In mesh networks with wormhole switching and virtual channels, we showed that good on-line message-scheduling policies and flit-multiplexing methods could greatly enhance network performance. Also, a new source of deadlock was found in our study of priority-based flit-multiplexing methods.

- When communicating task modules are to be mapped onto a distributed/parallel system, the underlying flow-control mechanisms were taken into account for our optimization problem. We emphasized the importance of choosing effective cost functions rather than the optimization algorithms themselves. With proper choices of cost functions, task-mapping strategies and flow-control mechanisms were found to be able to work in tandem.

- We studied the problem of mapping communicating subcubes into a binary hypercube. Properties of this type of mapping were investigated and mapping strategies were proposed. Previous research on subcube mapping has been mostly focused on independent subcubes without considering inter-subcube communication.

7.3 Future Work

There are several research topics related to this dissertation work that warrant further investigation. First, in Chapters 3 and 5, we only considered networks of which virtual-channel buffers were one flit long. When the buffer size is increased, how the network performance will be affected is an interesting issue. Although this has been addressed to some degree in [27], the author used a simplified model which ignored the overhead of entering or retrieving flits in larger buffers. For a more rigorous study, it is necessary to construct a more complex simulation program taking this overhead into account.

Also, we only considered mesh networks implemented with the non-adaptive e-cube routing algorithm. It should be interesting to investigate the problem in the context of adaptive routing algorithms such as those proposed in [32, 44]. We have shown that the SRBF message-scheduling policy and the SRBP flit-multiplexing method to be very effective when the $e-$cube algorithm is used. However, when another routing algorithm is used instead, the resulting performance characteristics are still unknown.

In Chapter 4, we only focused on low-complexity cost functions. It may be possible to propose a more complex cost function which requires more computation at each trial, but can reach a good mapping with less number of trials. Also, we only used the simulated annealing method to evaluate these cost functions. The possibility of using other methods like genetic algorithms should also be studied.

In Chapter 6, we showed that when the task graph is a star, an optimal parallel mapping is also optimal among all mappings. It is our conjecture that this is also true for general task graphs. A proof of this conjecture can greatly simplify the optimization problem.

Finally, in our simulations, the communication patterns used were mostly randomly generated. It will be interesting to evaluate the various schemes using actual program traces.

BIBLIOGRAPHY

[1] A. Agarwal, "Limits on interconnection network performance," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 398–412, October 1991.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

[3] A. Al-Dhelaan and B. Bose, "A new strategy for processor allocation in an n-cube multiprocessor," in *Proc. Intl. Phoenix Conf. Comput. Commun.*, pp. 114–118, March 1989.

[4] S. Anderson and M. C. Chen, "Parallel branch-and-bound algorithms on the hypercube," in *Hypercube Multiprocessors*, pp. 309–317, 1987.

[5] H. G. Badr and S. Podar, "An optimal shortest-path routing policy for network computers with regular mesh-connected topologies," *IEEE Trans. on Computers*, vol. 38, no. 10, pp. 1362–1370, October 1989.

[6] J. Baxter and J. Patel, "The LAST algorithm: A heuristic based static task allocation algorithm," in *Proc. of the International Conference on Parallel Processing*, pp. 217–222, 1989.

[7] F. Berman and L. Snyder, "On mapping parallel algorithms into parallel architectures," *Journal of Parallel and Distributed Computing*, vol. 4, no. 5, pp. 439–458, 1987.

[8] P. Berman, L. Gravano, G. Pifarre, and J. Sanz, "Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks," in *Symposium on Parallel Algorithms and Architectures*, pp. 3–12, June 1992.

[9] S. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed computer system," *IEEE Trans. on Software Engineering*, vol. 7, no. 6, pp. 583–589, 1981.

[10] K. Bolding, S.-C. Cheung, S.-E. Choi, C. Ebeling, S. Hassoun, T. A. Ngo, and R. Wille, "The Chaos router chip: Design and implementation of an adaptive router," in *Proc. of the International Conference on VLSI*, 1993.

[11] K. Bolding and L. Snyder, "Mesh and torus chaotic routing," in *Proc. Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems*, pp. 333–347, 1992.

[12] R. Boppana and S. Chalasani, "A comparison of adaptive wormhole routing algorithms," in *Proc. International Symposium on Computer Architecture*, pp. 351–360, 1993.

[13] G. Brassard and P. Bratley, *Algorithmics: Theory and Practice*, Prentice-Hall, 1988.

[14] J. Bruno, E. C. Jr., and R. Sethi, "Scheduling independent tasks to reduce mean finishing time," *Journal of the ACM*, vol. 17, no. 7, pp. 382–387, 1974.

[15] S.-G. Chang, "Fair integration of routing and flow control in communication networks," *IEEE Trans. on Communications*, vol. 40, no. 4, pp. 821–834, April 1992.

[16] M. S. Chen, *Distributed Routing and Task Allocation in Multicomputer Systems*, PhD thesis, The University of Michigan, 1988.

[17] M. S. Chen and K. G. Shin, "Processor allocation in an n-cube multiprocessor using gray codes," *IEEE Trans. on Computers*, vol. C-36, no. 12, pp. 1396–1407, December 1987.

[18] M. S. Chen and K. G. Shin, "Depth-first search approach for fault-tolerant routing in hypercube multicomputers," *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 152–159, April 1990.

[19] N. Chen and C. Liu, "On a class of scheduling algorithms for multiprocessor computing systems," in *Lecture Notes in Computer Science*, Springer, 1975.

[20] A. A. Chien and J. H. Kim, "Planar-adaptive routing: Low-cost adaptive networks for multiprocessors," in *Proc. International Symposium on Computer Architecture*, pp. 268–277, May 1992.

[21] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task allocation in distributed data processing," *Computer*, pp. 57–69, November 1980.

[22] R. Cypher and L. Gravano, "Adaptive, deadlock-free packet routing in multicomputer networks using virtual channels," in *Proc. of the International Conference on Parallel Processing*, pp. III–204–III–211, August 1992.

[23] W. Dally and H. Aoki, "Deadlock-free adaptive routing in multicomputer networks using virtual channels," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 4, pp. 466–475, April 1993.

[24] W. J. Dally, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. on Computers*, vol. C-36, no. 5, pp. 547–553, May 1987.

[25] W. J. Dally, "Performance analysis of k-ary n-cube interconnection networks," *IEEE Trans. on Computers*, vol. 39, no. 6, pp. 775–785, June 1990.

[26] W. J. Dally, "Express cubes: Improving the performance of k-ary n-cube interconnection networks," *IEEE Trans. on Computers*, vol. 40, no. 9, pp. 1016–1023, September 1991.

[27] W. J. Dally, "Virtual-channel flow control," *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 194–205, March 1992.

[28] W. J. Dally, J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler, "The message-driven processor: A multicomputer processing node with efficient mechanisms," *IEEE Micro*, vol. 12, no. 2, pp. 23–39, April 1992.

[29] L. Desbat and D. Trystram, "Implementing the discrete Fourier Transform on a hypercube vector-parallel computer," in *Proc. of the 4th Distributed Memory Computing Conference*, pp. 407–410, March 1989.

[30] D. Dolev, E. Upfal, and M. Warmuth, "The parallel complexity of scheduling with precedence constraints," *Journal of Parallel and Distributed Computing*, vol. 3, no. 4, pp. 553–576, 1986.

[31] J. W. Dolter, S. Daniel, A. Mehra, J. Rexford, W.-C. Feng, and K. G. Shin, "SPIDER: Flexible and efficient communication support for point-to-point distributed systems," in *Proc. of the 14-th Int'l Conf. Distributed Computing Systems*, pp. 574–580, June 1994.

[32] J. Duato, "Deadlock-free adaptive routing algorithms for multicomputers: Evaluation of a new algorithm," in *Proc. of the IEEE Symposium on Parallel and Distributed Processing*, pp. 840–847, 1991.

[33] S. Dutt and J. P. Hayes, "On allocating subcubes in a hypercube multiprocessor," in *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, pp. 801–810, January 1988.

[34] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Computer*, vol. 15, no. 6, pp. 50–56, June 1982.

[35] F. Ercal, J. Ramanujam, and P. Sadayappan, "Task allocation onto a hypercube by recursive min-cut bipartitioning," in *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, pp. 210–221, January 1988.

[36] S. Felperin, L. Gravano, G. Pirarre, and J. Sanz, "Fully-adaptive routing: Packet switching performance and wormhole algorithms," in *Supercomputing*, pp. 654–663, November 1991.

[37] S. Felperin, L. Gravano, G. Pirarre, and J. Sanz, "Routing techniques for massively parallel communication," *Proceedings of the IEEE*, vol. 79, no. 4, pp. 488–503, April 1991.

[38] G. Fox, "Parallel computing comes of age," *Concur. Pract. Exp.*, vol. 1, no. 1, pp. 63–103, 1989.

[39] H. Gabow, "Scheduling UET systems on two uniform processors and length two pipelines," *SIAM J. Comput.*, vol. 17, no. 4, pp. 810–811, 1988.

[40] M. Garey and D. Johnson, "Complexity results for multiprocessor scheduling with resource constraints," *SIAM J. Comput.*, vol. 4, no. 4, pp. 396–411, 1975.

[41] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Co., 1979.

[42] P. T. Gaughan and S. Yalamanchili, "Adaptive routing protocols for hypercube interconnection networks," *IEEE Computer Magazine*, vol. 26, no. 2, pp. 12–23, May 1993.

[43] P. T. Gaughan and S. Yalamanchili, "Analytical models of bandwidth allocation in pipelined k-ary n-cubes," Technical Report TR-GIT/CSRL-93/03, School of Electrical Engineering, Georgia Institute of Technology, 1993.

[44] C. J. Glass and L. M. Ni, "The turn model for adaptive routing," in *Proc. of the 19th International Symposium on Computer Architecture*, pp. 278–287, May 1992.

[45] J. M. Gordon, *Efficient schemes for massively fault-tolerant parallel communications*, PhD thesis, The University of Michigan, 1990.

[46] J. M. Gordon and Q. F. Stout, "Hypercube message routing in the presence of faults," in *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, pp. 318–327, January 1988.

[47] R. Graham, "Bounds on multiprocessing timing anomalies," *SIAM J. Appl. Math.*, vol. 17, no. 2, pp. 416–429, 1969.

[48] A. Greenberg, "Deflection routing in hypercube networks," *IEEE Trans. on Communications*, vol. 40, no. 6, pp. 1070–1081, 1992.

[49] D. Greenberg and S. Bhatt, "Routing multiple paths in hypercubes," in *Symposium on Parallel Algorithms and Architectures*, pp. 45–54, July 1990.

[50] D. Gusfield, "Parametric combinatorial computing and a problem of program module distribution," *Journal of the ACM*, vol. 30, no. 3, pp. 551–563, 1983.

[51] D. Helmbold and E. Mayr, "Two processor scheduling is in NC," *SIAM J. Comput.*, vol. 16, no. 4, pp. 747–759, 1987.

[52] M. Herbordt, C. Weems, and J. Corbett, "Message passing algorithms for an SIMD torus with coteries," in *Symposium on Parallel Algorithms and Architectures*, pp. 11–20, July 1990.

[53] C.-T. Ho and S. L. Johnson, "Distributed routing algorithms for broadcasting and personalized communications in hypercubes," in *Proc. of the 1986 International Conference on Parallel Processing*, pp. 640–648, August 1986.

[54] C.-T. Ho and S. L. Johnson, "Algorithms for matrix transposition on boolean n-cube configured ensemble architectures," in *Proc. of the 1987 International Conference on Parallel Processing*, pp. 621–629, 1987.

[55] S. Horiike, "A task mapping method for a hypercube by combining subcubes," in *Proc. of the Fifth Distributed Memory Computing Conference*, pp. 909–914, April 1990.

[56] T.-W. Hou, S. Tsai, and L. Tseng, "Adaptive and fault-tolerant routing algorithms for high performance 2D torus interconnection network," *Computers Math. Applic.*, vol. 23, no. 1, pp. 3–15, 1992.

[57] C. E. Houstis, "Allocation of real-time applications to distributed systems," in *Proc. of the 1987 Int'l Conf. on Parallel Processing*, pp. 863–866, August 1987.

[58] T. Hu, "Parallel sequencing and assembly line problems," *Oper. Research*, vol. 9, pp. 841–848, 1961.

[59] J.-J. Hwang, Y.-C. Chow, F. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM J. Comput.*, vol. 18, no. 2, pp. 244–257, 1989.

[60] C. R. Jesshope, P. R. Miller, and J. T. Yantchev, "High performance communications in processor networks," in *Proc. of the International Symposium on Computer Architecture*, pp. 150–157, May 1989.

[61] S. L. Johnson, "Communication efficient basic linear algebra computations on hypercube architectures," *Journal of Parallel and Distributed Computing*, vol. 4, pp. 133–172, 1987.

[62] S. Johnsson, "Communication in network architectures," in *VLSI and Parallel Computation*, pp. 223–378, Morgan Kaufmann, 1990.

[63] E. C. Jr., M. Garey, and D. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM J. Comput.*, vol. 7, no. 1, pp. 1–17, 1978.

[64] H. Jung, L. Kirousis, and P. Spirakis, "Lower bounds and efficient algorithms for multiprocessor scheduling of dags with communication delays," in *Proc. of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 254–264, 1989.

[65] D. Kafura and V. Shen, "Task scheduling on a multiprocessor system with independent memories," *SIAM J. Comput.*, vol. 6, no. 3, pp. 568–597, 1989.

[66] D. D. Kandlur and K. G. Shin, "Traffic routing for multicomputer networks with virtual cut-through capability," *IEEE Trans. on Computers*, vol. 41, no. 10, pp. 1257–1270, October 1992.

[67] M. Kaufmann, "An almost-optimal algorithm for the assembly line scheduling problem," *IEEE Trans. on Computers*, vol. 23, no. 11, pp. 1169–1174, November 1974.

[68] C. K. Kim and D. A. Reed, "Adaptive packet routing in a hypercube," in *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, pp. 625–630, January 1988.

[69] J. Kim, C. R. Das, and W. Lin, "A top-down processor allocation scheme for hypercube computers," *IEEE Trans. on Parallel and Distributed Systems*, vol. 2, pp. 20–30, January 1991.

[70] D. L. Kiskis and K. G. Shin, "Embedding triple-modular redundancy into a hypercube architecture," in *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, pp. 337–345, January 1988.

[71] D. W. Krumme, K. N. Venkataraman, and G. Cybenko, "Hypercube embedding is NP-complete," in *Proc. of the First Conf. on Hypercube Concurrent Computers and Applications*, pp. 148–157, August 1985.

[72] P. J. M. Laarhoven and E. H. L. Aarts, *Simulated Annealing: Theory and Applications*, D. Reidel Publishing Company, 1987.

[73] C.-Y. Lee, J.-J. Hwang, Y.-C. Chow, and F. Anger, "Multiprocessor scheduling with interprocessor communication delays," *Oper. Res. Lett.*, vol. 7, no. 3, pp. 141–145, 1988.

[74] S. Lee and J. Aggarwal, "A mapping strategy for parallel computing," *IEEE Trans. on Computers*, vol. 36, no. 4, pp. 433–442, April 1987.

[75] F. Leighton, *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann, 1992.

[76] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, 1981.

[77] J. Lindberg and S. Yalamanchili, "Static program assignment in circuit switched multiprocessors," in *Proc. 6-th Distributed Memory Computing Conference*, pp. 244–247, April 1991.

[78] V. M. Lo, "Task assignment to minimize completion time," in *Proc. of the Fifth Int'l Conf. on Distributed Computer Systems*, pp. 329–336, May 1985.

[79] V. M. Lo, "Temporal communication graphs: A new graph theoretic model for mapping and scheduling in distributed memory systems," in *Proc. 6-th Distributed Memory Computing Conference*, pp. 248–252, April 1991.

[80] D. Martin and G. Estrin, "Experiments on models of computation and systems," *IEEE Trans. on Elect. Comput.*, vol. 16, no. 1, pp. 59–69, 1967.

[81] J. J. Metzner, "Message scheduling for efficient data communication under varying channel conditions," *IEEE Trans. on Communications*, vol. 32, no. 1, pp. 48–55, January 1984.

[82] D. Mitra and R. Cieslak, "Randomized parallel communications," in *Proc. of the 1986 International Conference on Parallel Processing*, pp. 224–230, August 1986.

[83] J. Ngai and C. Seitz, "A framework for adaptive routing in multicomputer networks," in *Symposium on Parallel Algorithms and Architectures*, pp. 1–9, June 1989.

[84] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The J-Machine multicomputer: An architectural evaluation," in *Proc. International Symposium on Computer Architecture*, pp. 224–235, 1993.

[85] M. G. Norman, "Models of machines and computation for mapping in multicomputers," *ACM Computing Surveys*, vol. 25, no. 3, pp. 263–302, September 1993.

[86] S. Padmanabhan and C. Baru, "Routing between subcubes in a hypercube," in *Proc. of the 6th Distributed Memory Computing Conference*, pp. 295–298, April 1991.

[87] C. H. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," *SIAM J. Comput.*, vol. 19, no. 2, pp. 322–328, 1990.

[88] Y. Park, V. Cherkassky, and G. Lee, "ATM cell scheduling for broadband switching systems by neural networks," in *Proc. Int'l Workshop Application Neural Networks to Telecommunications*, pp. 112–118, 1993.

[89] V. G. J. Peris, M. S. Squillante, and V. K. Naik, "Analysis of the impact of memory in distributed parallel processing systems," in *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 5–18, May 1994.

[90] M. Pertel, "A critique of adaptive routing," Technical Report CS-TR-92-06, California Institute of Technology, 1992.

[91] D. Pritchard, C. Askew, D. Carpenter, I. Glendinning, A. Hey, and D. Nicole, "Practical parallelism using transputer arrays," in *Parallel Architectures and Languages*, Springer-Verlag, 1987. Lecture Notes in Computer Science.

[92] M. J. Quinn, "Implementing best-first branch-and-bound algorithms on hypercube multicomputers," in *Hypercube Multiprocessors*, pp. 318–326, 1987.

[93] S. Raghupathy, M. R. Leuze, and S. R. Schach, "Message routing schemes in a hypercube machine," in *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, pp. 640–646, January 1988.

[94] G. Rao, H. Stone, and T. Hu, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Trans. on Computers*, vol. 28, no. 4, pp. 291–298, April 1979.

[95] V. Rayward-Smith, "UET scheduling with unit interprocessor communication delays," *Disc. Appl. Math.*, vol. 18, no. 1, pp. 55–71, 1987.

[96] C. L. Seitz, "The cosmic cube," *CACM*, vol. 28, pp. 22–23, January 1985.

[97] D. D. Sharma and D. K. Pradhan, "A novel approach for subcubes allocation in hypercube multiprocessors," in *Proc. of the Fourth IEEE Intl. Symposium on Parallel and Distributed Processing*, pp. 336–345, 1992.

[98] C. Shen and W. Tsai, "A graph matching approach to optimal task assignment in distributed computing system using a minimax criterion," *IEEE Trans. on Computers*, vol. c-34, no. 3, pp. 197–203, March 1985.

[99] K. G. Shin and J. W. Dolter, "An alternative majority voting method for real-time computing systems," *IEEE Trans. on Reliability*, vol. 38, no. 1, pp. 58–64, April 1989.

[100] B. Shirazi, M. Wang, and G. Pathac, "Analysis and evaluation of heuristic methods for static task scheduling," *Journal of Parallel and Distributed Computing*, vol. 10, no. 3, pp. 222–232, 1990.

[101] H. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. on Software Engineering*, vol. 3, no. 1, pp. 85–93, January 1977.

[102] H. Stone, "Program assignment in three processor systems and tricutset partitioning of graphs," Technical Report ECE-CS-77-7, University of Massachusetts, 1977.

[103] Q. Stout and B. Wagar, "Intensive hypercube communication–prearranged communication in link-bound machines," *Journal of Parallel and Distributed Computing*, vol. 10, no. 2, pp. 167–181, 1990.

[104] T. Tang, "Parallel sorting on the hypercube concurrent processor," in *Proc. of the 5th Distributed Memory Computing Conference*, pp. 237–240, April 1990.

[105] R. Thurimella and Y. Yesha, "A scheduling principle for precedence graphs with communication delay," in *Proc. of the International Conference on Parallel Processing*, pp. III–229–III–236, August 1992.

[106] D. Towsley, "Allocating programs containing branches and loops within a multiple processor system," *IEEE Trans. on Software Engineering*, vol. 12, no. 10, pp. 1018–1024, October 1986.

[107] B.-R. Tsai and K. G. Shin, "Communication–oriented assignment of task modules in hypercube multicomputers," in *Proc. 12–th Int'l Conf. on Distributed Comput. Syst.*, pp. 38–45, June 1992.

[108] B.-R. Tsai and K. G. Shin, "Combined routing and scheduling of concurrent communication traffic in hypercube multicomputers," submitted to publication, 1993.

[109] B.-R. Tsai and K. G. Shin, "Mapping concurrent communicating modules to mesh multicomputers equipped with virtual channels," submitted to publication, 1994.

[110] L. Valiant, "Universal schemes for parallel communication," in *Proc. of the 13th ACM Symposium of Theory of Computing*, pp. 263–277, May 1981.

[111] L. Valiant, "A scheme for fast parallel communication," *SIAM J. Comput.*, vol. 11, no. 2, pp. 350–361, 1982.

[112] S. Vassilopoulos and P. Papantoni-Kazakos, "A transmission scheduling algorithm for mixed traffic: High and low priority," in *Proc. IEEE INFOCOM*, pp. 2251–2259, 1992.

[113] B. Wagar and Q. F. Stout, "Passing messages in link-bound hypercubes," in *Hypercube Multiprocessors*, M. Heath, editor, pp. 251–257, SIAM, 1987.

[114] A. Wagner and D. G. Corneil, "Embedding trees in a hypercube is NP–complete," *SIAM J. Comput.*, vol. 19, no. 4, pp. 570–590, 1990.

[115] E. Williams, "Assigning processes to processors in distributed systems," in *Proc. of the IEEE Conference on Parallel Processing*, pp. 404–406, 1983.

[116] T.-K. Woo, "A decentralized scheduling algorithm for distributed systems," in *Proc. IEEE SOUTHEASTCON*, pp. 205–208, 1991.