MAPPING PARALLEL ALGORITHMS TO MESSAGE PASSING MACHINES

by

XIAOXIONG ZHONG

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 1994

"Mapping Parallel Algorithms to Message Passing Machines," a dissertation prepared by Xiaoxiong Zhong in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

Virginia M. Lo
Chair of the Examining Committee

5/28/94
Date

Commitee in charge:
Dr. Virginia Lo, Chair
Dr. Sanjay Rajopadhye
Dr. Andrzej Proskurowski
Dr. Evan Tick
Dr. Brad Shelton

Accepted by:
Vice Provost and Dean of the Graduate School

An Abstract of the Dissertation of

Xiaoxiong Zhong for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken June 1994

Title: MAPPING PARALLEL ALGORITHMS TO MESSAGE PASSING MACHINES

Approved: Virginia M. Lo

Dr. Virginia Lo

Message passing machines provide an opportunity to achieve high performance for applications such as those in scientific computing, in digital signal processing, in simulation and in electronic design automation. Programming such architectures to achieve high performance, however, poses a challenging task to users. One of the major problems is to design a mapping (assignment) scheme for processes to deal with the mismatch between the ideal communication structure for the parallel algorithm and the target architecture. The research in this thesis aims at this problem. In the thesis, two kinds of architectures, multicomputers and systolic arrays, are considered.

In the first part, we concentrate on communication issues in the design of mapping algorithms for a multicomputer. As communication switching technologies advance, major factors which incur communication overhead have changed and should be studied. We empirically study the effect of communication overhead caused by the topological mismatch in a multicomputer Empirical case studies are carried out to qualitatively characterize the impact of several important mapping metrics and

architectural factors on the performance of benchmarks. To quantify communication overhead, we study and validate analytical estimation formulae for message latency. The applications of the message latency formulae to a general purpose multicomputer simulator are discussed and a new parallel program performance evaluation framework is proposed.

We then study the problem of reducing communication overhead by utilizing knowledge of the message passing requirement in an application. We propose efficient application-specific routing algorithms to reduce communication overhead. The proposed techniques can be applied to parallel programs with intensive communication on a multicomputer with user-controlled routing capability.

In the second part, we study the problem of mapping a class of algorithms called regular iterative algorithms to systolic arrays. First, the problem of finding an optimal time schedule for regular iterative algorithms is studied. Second, we propose a systematic method to enumerate linear allocation functions to yield spatially regular systolic arrays with some permissible connection constraints. Such a method can be used to design systolic arrays based on various optimization criteria. Finally, we show how to improve efficiency of a systolic array to an almost 100% efficient array by using a quasi-linear transformation function. The results have potential applications in a high level synthesis system or in a silicon compiler.

CURRICULUM VITA

NAME OF THE AUTHOR: Xiaoxiong Zhong

PLACE OF BIRTH: Guangdong, China

DATE OF BIRTH: November 27, 1963

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Fudan University, China

DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science, 1994, University of Oregon

Master of Science in Computer Science, 1986, Fudan University, China

Bachelor of Science in Computer Science, 1983, Fudan University, China

PROFESSIONAL EXPERIENCE:

Lecturer, Department of Computer Science, Zhongshan University, Guangdong, China, 1986-1989

Graduate Teaching and Research Fellow, Department of Computer and Information Science, University of Oregon, Eugene, 1989-1993

Senior Software Engineer, Zycad Corporation, Fremont, California, 1993-1994

ACKNOWLEDGEMENTS

I wish to express my deep appreciation and thanks to my advisors Dr. Ginnie Lo and Dr. Sanjay Rajopadhye for their continuous support and encouragement. The guidance, care and understanding from Ginnie have always been a major source of strength for me. Working with Sanjay has been stimulating. I am grateful to Dr. Evan Tick for his help, motivation, and advice. His detailed corrections and comments greatly enhanced the quality of this thesis. Dr. Andrzej Proskurowski has been very supportive during my stay at the University of Oregon and I am very thankful for all his help. I thank Dr. Brad Shelton for taking time to be on my committee.

I am also grateful for the support from Zycad Corporation. The encouragement my manager Ramesh provided me in the last few months has been very helpful in finishing my dissertation.

I thank my graduate fellows A.V.S.Sastry, Jan Telle, and many others for their help. I thank our graduate secretary Betty Lockwood.

My friend Norman Lumian has made my stay in Eugene enjoyable and interesting.

The care and understanding from my parents and sisters in China have been important to me. My wife Wensheng has always been behind me during this long and sometimes difficult journey. This would not have been possible without her understanding and support.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Page

CHAPTER I

INTRODUCTION

Message passing machines are designed for scalable high performance computing. Recent developments in computer architecture as well as communication technology have shown that such machines are highly promising for a broad range of application areas. A message passing machine differs from other parallel architectures in that processes in different processors communicate with each other solely via message passing. The key features of a message passing machine are:

1. Processors are physically connected by a point to point network (i.e., *direct* network). Example networks are hypercubes, meshes and fat-trees.

2. Each processor element has its own local memory. No shared memory exists.

3. A message passing machine usually has a large number of processors. It can be scaled up to hundreds or thousands of processors.

In this thesis, we concentrate on two kinds of architectures: multicomputers and systolic arrays. Multicomputers are general purpose machines which are best for medium and coarse grain computations, while systolic arrays are application-specific machines which are well suited to fine-grain applications exhibiting massive parallelism. Although the two architectures are different, recent developments have led to the evolution of several machines which represent a hybrid between a multicomputer and a systolic array, as evidenced by the Intel iWarp system [14, 49].

Although the development in hardware of such machines has advanced dramatically in the last few years, software support for these machines has lagged far behind, which is the biggest hindrance to the widespread use of such machines. One of the major difficulties in programming such machines is the *mapping* problem. The mapping problem involves the mismatch that usually exists between the underlying ideal computation and communication structures of an application and the resources in a target architecture. This is a very important, challenging, cumbersome and error-prone process. It is important because it is vital to performance: a careless mapping may result in poor load balancing and large communication overhead. It is challenging because designing a good mapping may require detailed understanding of scheduling, graph theory, combinatorics and the properties of target architectures. It is also cumbersome since the mapping problem is an extra burden for the programmer in addition to the programming task. Finally, the task of developing a mapping is error-prone due to its complicated nature.

The Mapping Problem

The performance of a program on a message passing machine is determined by two factors: the inherent parallelism in the program and the overhead incurred by the target architecture to achieve the inherent parallelism. The first factor is determined by the algorithm used in the program. For example, the maximum number of processes which can be executed at the same time may be limited. The second factor is determined by the degree of conflict between the demand for resources required from the program and the available resources of the target machine. In this thesis, we concentrate on the second problem.

The mapping problem [8, 80] can be described as the problem of assigning tasks

to processors and and assigning messages to paths in the target architecture. The programming paradigm used in this research belongs to the communicating sequential process paradigm [52]. More precisely, in this model, a parallel application consists of a set of sequential processes. Processes communicate with each other with explicit message passing. Many programming languages for message passing machines fall into this class [108, 82]. For example, the C programming language supported for the Intel Paragon is a standard C language with extensions for explicit message passing. Other languages or language interfaces which belong to this model include DINO [101] and PICL [42]. Recently, some specialized languages have been designed for the purpose of mapping [9, 81, 7, 8]. Such languages allow a mapping tool to effectively extract useful information needed in a mapping algorithm.

In general, two types of mismatch exist between a parallel application and a target message passing machine [8]. The first is due to the mismatch between the number of processors available in the target architecture and the number of processes in the application. Figure 1 shows an example of an application which is modeled as a complete binary tree and a mesh-connected target architecture. In this example, the number of processes is 16 but there are only 6 processors. To deal with this mismatch, one should be aware of the following factors. First, it is not always true that using the same number of processors as that of processes renders the maximum speedup. This is because communication overhead may seriously degrade the whole computation if the computation grain is too small. Second, if the number of available processors of the target machine is less than that of processes, several processes are forced to be clustered onto one physical processor. How to achieve load balancing and minimize communication overhead in order to maximize performance (or minimize

total completion time) is a difficult problem. Third, even though one is given a sufficient number of processors, some of them may be wasted due to the effect of synchronization. For example, a processor may be idle waiting for messages to arrive. This introduces the concept of efficiency (processor utilization) and a good mapping should try to maximize processor utilization. This problem is related to process scheduling.

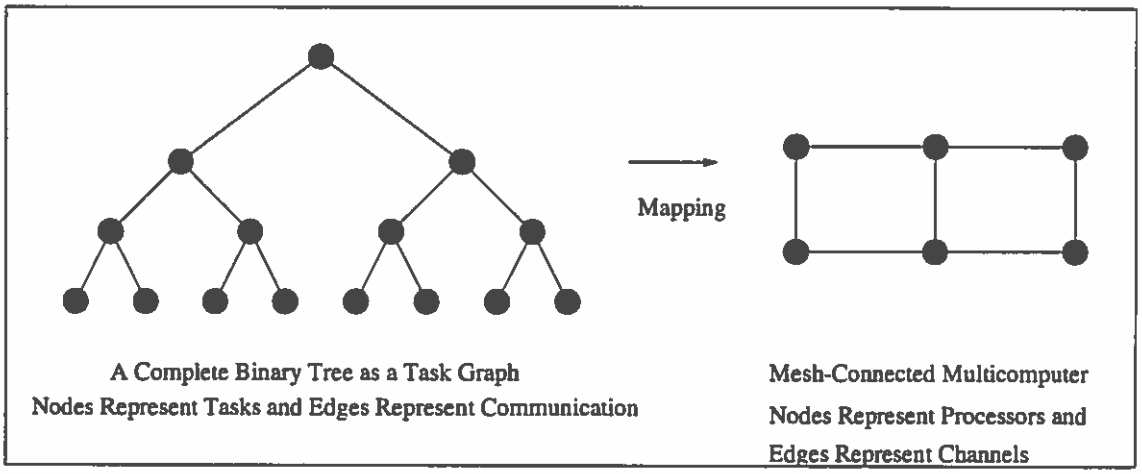


Figure 1: An Example of Mismatch Between an Application and a Target Architecture.

The second type of mismatch is due to the difference between the ideal interconnection topology used in the application and the communication network topology of the target architecture. Such a mismatch causes communication overhead since messages are delayed due to long message paths and traffic congestion in the network. To cope with this type of mismatch, one should carefully assign processes to processors so that the topological mismatch is minimized based on various criteria. This step is called process placement. Different communication technologies may cause different communication overhead for the same placement strategy. Thus, the metrics used in

this step should be sensitive to the communication technology used. Furthermore, if the target architecture allows the user to control routing, one can further design an efficient routing for the specific application to reduce traffic congestion after processes have been assigned to processors.

To summarize, a message passing machine can be viewed as a system consisting of two types of resources: processors and communication channels. The mapping problem arises from the conflicting demands for these resources. The first mismatch is due to the conflicting demand for processors and the second mismatch is due to the conflicting demand for communication channels.

Problems Addressed in the Dissertation

In this dissertation, we study the problem of mapping parallel algorithms to two kinds of architectures, namely, multicomputers and systolic arrays. The thesis correspondingly consists of two parts.

Part I: Communication Issues in Mapping to Multicomputers

Reducing the topological mismatch in mapping parallel algorithms to a multicomputer has been one of the most important problems for the performance of an application in old generation multicomputers such as Intel's iPSC/1 and Caltech Cosmic Cube [40, 5, 8]. As the communication technologies used in a multicomputer advance, the communication overhead caused by the topological mismatch has been reduced dramatically. Such a change calls for a re-examination of the performance effect of the communication overhead for the new communication technologies. Specifically, we address three fundamental questions to better understand the nature of the communication overhead caused by the topological mismatch.

- *Is the communication overhead caused by the topological mismatch stiill critical to performance under the new communication technologies?* For old generation multicomputers which use the store-forward routing scheme, the communication overhead due to the topological mismatch was vital to mapping performance. For new generation multicomputers which use the wormhole routing scheme, however, there has been a claim that the communication overhead may no longer seriously affect performance. Other evidence indicates that the communication overhead continues to be important. A more careful investigation is necessary to resolve this issue.

- *What contributes to the communication overhead?* In the old generation multicomputers with store-forward routing, key factors such as dilation have been used in the development of mapping algorithms. For the new wormhole routing scheme, we need to develop new metrics to better characterize the communication overhead of a mapping.

- *How to reduce communication overhead when designing a mapping algorithm?* We need to develop new mapping algorithms to reduce communication overhead based on the new metrics.

Part I of the thesis focuses on the above issues.

Part II: Mapping to Systolic Arrays

In Part II, we consider the problem of designing systolic arrays starting from a high-level algorithm expressed as a class of programs called *Regular Iterative Algorithms* (RIAs). Based on well known techniques, the first step in designing a systolic array for RIAs is to schedule computations based on the dependencies in the RIAs,

and the second step is to assign computations of the RIA to physical processors, the so called processor allocation (assignment) problem.

In practice, a systolic array is designed based on various optimization criteria. Among many design criteria [68], the important ones include total completion time, number of processors, and processor utilization. Correspondingly, we address the following three problems.

- *Optimal timing schedule to minimize execution time:* To minimize execution time for each computation, one would desire to schedule the computation as early as possible with the constraint that the dependencies are not violated. To achieve this, we should investigate the nature of the optimal schedule and how to derive such a schedule.

- *Processor allocation based on various design criteria:* Many design criteria such as the minimal number of processors are based on the way that processors are allocated for computations. It is thus important to study methods to derive processor allocation schemes.

- *Processor efficiency:* In the execution of the application, processors in the derived array may be only actively doing useful work at some time and are idle for the rest of time. It is thus very important to derive an array with maximum processor utilization during the execution of the application without sacrificing other optimization design criteria such as total completion time and processor complexity.

Overview of the Dissertation

In the first part, we concentrate on communication issues in the mapping problem. We empirically study the effect of communication overhead caused by the topological mismatch in a multicomputer under two kinds of communication technology, namely, the store-forward and the wormhole routing schemes. Empirical case studies are carried out to qualitatively characterize the impact of several important mapping metrics and architectural factors on the performance of benchmarks. We show that the communication overhead caused by the topological mismatch can still significantly affect the performance of a benchmark. To quantify communication overhead, we propose and validate analytical estimation formulae for message latency. The formulae take runtime contention information into account and are directly sensitive to an application. We discuss incorporation of the message latency formulae in a general purpose multicomputer simulator, and a new parallel program performance evaluation framework that uses the formulae is proposed.

We then study the problem of reducing communication overhead by utilizing knowledge of the message passing requirements in an application. We propose efficient application-specific routing algorithms to reduce communication overhead based on the newly developed metrics. The proposed techniques can be applied to parallel programs with intensive communication in a multicomputer with user-controlled routing capability such as Intel's iWarp systems or Meiko's transputer systems.

In the second part, the problem of mapping a class of algorithms called *R*egular *I*terative *A*lgorithms (RIAs) to a systolic array is studied. To achieve the inherent parallelism exhibited in a RIA, the problem of finding the optimal timing schedule is studied. To map such an algorithm to a systolic array which has constraints on its

physical connections, a systematic method to enumerate linear allocation functions is developed. Such a method can be used to design systolic arrays based on various optimization criteria. To fully utilize processor resources, we study the problem of deriving a fully efficient systolic array with respect to processor utilization. A systematic method is proposed to generate a 100% efficient array without slowing down the array and without adding extra functional units. Our work solves the problem of mapping RIAs to systolic arrays in three important aspects, namely, timing schedule, processor allocation, and processor utilization.

Part I

COMMUNICATION ISSUES IN MAPPING TO MULTICOMPUTERS

CHAPTER II

MAPPING TO MULTICOMPUTERS

From the store-forward flow control scheme in old generation multicomputers to the wormhole and circuit-switching schemes used in current advanced multicomputers, the transition of communication switching technologies has made communication orders of magnitude faster. These changes entail a careful re-examination of classic methods and metrics which have been used in designing a mapping tailored to an old generation multicomputer. Empirical studies should be conducted to reveal the nature of the communication overhead incurred by these new technologies and their effects on the performance of an application should be characterized. New mapping algorithms and metrics to reduce communication overhead in a mapping should be developed. This part of the thesis focuses on the above problems.

In.this chapter, we describe the background knowledge needed to understand the communication issues related to the mapping problem. A general framework for mapping to a multicomputer is described. Several communication switching techniques used in multicomputers are introduced and different routing schemes are described. The deadlock problem, which is more critical in a wormhole routing scheme, is also introduced. We then identify three fundamental problems of communication overhead related to mapping and introduce communication overhead metrics which will be used throughout this part. Finally, we overview the work in Part I.

Introduction

As the technology of multicomputer architecture advances, major factors which affect the performance of a parallel program on a multicomputer have changed. In particular, new switching technologies used on a multicomputer have reduced communication overhead dramatically. This change introduces a new research area to address performance issues related to communication overhead and how to reduce this overhead in a mapping.

Specifically, to achieve good performance for a mapping, one has to understand how the performance of a parallel program is related to communication overhead incurred by specific communication technologies, what are the major factors which affect the communication overhead in terms of a mapping, how to measure these effects for the purpose of mapping, and what kinds of new techniques can be used to reduce communication overhead.

Early work on mapping did not take the network communication technology into account or only used a very simple model of the multicomputer network. For example, in the pioneering work on mapping, Stone [112], Bokhari [11], Lo [77] and other researchers [105] proposed load balancing schemes based on simple static approximations of communication overhead. In such a model, little information about the dynamic runtime behavior of the application is assumed. In the task scheduling area, a traditional directed acyclic graph based model [106, 91] also assumes statically determined communication costs among tasks, represented as the weights of edges. These models assume an all or nothing character to communication overhead based on whether a pair of communicating tasks were assigned to the same processor (zero overhead) or different processors (fixed overhead). Lewis and Rewini [36]

used a limited model of a store-forward multicomputer network to reflect the possible communication overhead for a schedule. In their model, the communication overhead between two tasks is not only determined by the message size but also is determined by the number of hops and the approximate network traffic congestion for this message.

We believe that it is important to develop new approaches to understanding the communication overhead of multicomputers and their relation to mapping. Our study must be sensitive to new communication technologies, specifically the wormhole routing scheme. We must develop a framework and new metrics for communication overhead. Furthermore, mapping techniques to reduce the overhead need to be carefully designed and evaluated.

A General Framework for Mapping to Multicomputers

This section describes a general framework for mapping to multicomputers. We describe characteristics of a multicomputer, the computation model used in mapping and general mapping approaches.

Multicomputers

A multicomputer is a general purpose message passing machine. Below, we discuss the characteristics of a multicomputer.

1. A multicomputer is an MIMD (multiple instruction, multiple data stream). Each processor has its own control unit and executes its own instruction stream.

2. A multicomputer usually consists of powerful computing engines (ranging from several to hundreds of MFLOPS) in each processor element.

3. Processors in a multicomputer are interconnected by a direct network. Examples of well-known multicomputer networks include a mesh, a tree, a hypercube and a torus.

4. A multicomputer has communication hardware responsible for message routing in each processor. Second generation multicomputers have dedicated communication modules separate from computation engines. Transmitting a message through an intermediate node does not need to interrupt the computation carried out by the local processor.

5. A multicomputer has moderate communication overhead. Second generation multicomputers support fast communication. For example, in an IPSC/2-SX system, sending a double-precision number to its neighboring processor only takes $3.0\mu s$, less than that of performing a floating point operation ($3.6\mu s$) [4].

6. Processors in a multicomputer execute asynchronously. No global clock is available.

Typical commercial multicomputers include Thinking Machine's CM-5, Intel's Paragon, iPSC/860, iPSC/2, iWarp and Ncube's Ncube/3200 and Ncube/2.

Computation Models

Several graph theoretic models of parallel computations have been designed and used in the mapping community. The *static task graph* [112, 11, 77, 105] statically represents a parallel application as an undirected weighted graph with nodes representing processes and edges representing communication between processes. Node weights are used to represent execution costs associated with the processes, and edge

weights are used to indicate the degree of communication between two processes. This has been one of the dominant models used in the parallel and distributed computing community for mapping. The model is simple and relatively easy to construct from the original program. One of the drawbacks of the model is that it does not capture the temporal behavior of the program, which may be vital to mapping.

The second predominant model is the *Directed Acyclic Graph* (DAG) [106, 91, 25, 3] model. In the DAG model, nodes represent tasks and directed edges represent a dependency relation (such as a message send/receive relation) between two tasks. Furthermore, a DAG can be weighted such that node weights represent the execution cost of tasks and edge weights represent size of messages between two tasks. The DAG model was originally developed for research in scheduling and has been used in parallelizing compilers that parallelize sequential code whose data and control graphs can be represented as DAGs. While the DAG model has been utilized in the mapping community, in a communicating sequential program, it fails to capture the identities of processes which persist over the lifetime of the computation.

Recently, a model called the *Temporal Communication Graph* (TCG) was proposed [78] to unify the two earlier models. In practice, many parallel applications proceed through computation and communication in a phase-by-phase fashion. Processes, however, in such a phase-by-phase execution, retain their identities during the lifetime of the program execution. The TCG model captures the process identity concept and also represents computation and communication phases explicitly. Compared with the static task graph model and the DAG model, a TCG represents both the topological and the temporal precedence information and is thus more general. A language called LaRCS has been designed specifically to describe the TCG

model [81]. In this thesis, we will use static task graphs as well as TCGs as our model of computation.

Overview of Mapping Techniques

The problem of optimally mapping a general static task graph, a DAG, or a TCG to target architectures with respect to many well-known metrics is difficult. Such optimization problems are usually NP-hard [41, 25]. To simplify the complicated process, researchers have tried to decompose the mapping problem into several steps. First, information useful to the mapping is extracted from the initial description of the application, based on the computation model chosen. Several languages have been designed to represent computation models. Such languages can be a programming language such as the one used in Prep-P [9] or a special language such as LaRCS used in Oregami [81, 9]. Second, contraction is performed to minimize the mismatch between the number of available processors and the number of processes. Third, clustered processes are assigned (placed) to processors and fourth, if the target architecture allows the user to control routing or switching setting, application-specific routing can be performed to reduce traffic congestion. In addition to the above steps, if temporal information is considered, as in the DAG or the TCG model, scheduling can be performed to further specify the execution order of the processes to enhance the processor utilization. Figure 2 illustrates the first four steps of a mapping.

Notice, however, that several steps may be combined into a single step and some steps may not be necessary. For example, to take advantage of the difference in computing power of nodes in a heterogeneous environment so that processes with heavy execution cost can be assigned to faster nodes, the contraction and placement steps may need to be combined into a single step so that processes contracted are

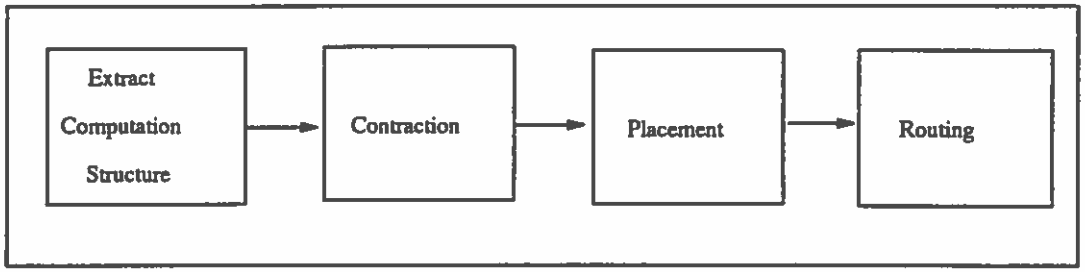


Figure 2: Illustration of a Mapping Procedure.

assigned (placed) to specific nodes.

The contraction step in the mapping problem is usually performed to minimize process-processor mismatch, to balance processor workload, and to minimize inter-processor communication. A careful tradeoff is necessary since these these goals may conflict with each other. For the static task graph model, graph-theoretic methods have been proposed. These include Stone's classic minimal-cut network flow approach [112], which later was extended by Lo, Bokhari and others [77, 11]. Some other heuristics such as nearest neighboring clustering method have also been developed [105, 70].

For the DAG model, Sarkar [106], Kim and Brown [60], Papadimitrious [89] and recently Yang and Gerasoulis [44, 45] have studied the problem of clustering processes in a DAG to minimize the total completion time under the assumption that the number of processors is infinite. The clustered graph (which may no longer be a DAG) may be further clustered by heuristics used in the static task model.

For the TCG model, information on the temporal behavior of the computation can be used to improve processor utilization by clustering several processes which do not have conflicting computation phases. For example, for divide and conquer algorithms, the original complete binary tree computation structure can be contracted,

resulting in a binomial tree, to improve efficiency [79, 127], or a specific clustering can be tailored to the target architecture [123]. For a general TCG, no systematic work has been done.

The placement step is performed to minimize the topological mismatch by placing processes in order to minimize various metrics including maximum dilation, which is defined as the maximum distance messages have to travel in the target architecture. A large body of work has been carried out to develop embeddings from a task graph to the target interconnection network [100, 56, 71]. Quadratic assignment and other heuristics [50, 70] have also been studied for this purpose.

The routing step involves routing messages through a path in the network once processes have been assigned to processors. While many machines provide fixed routing schemes that are controlled by the hardware, several advanced multicomputers such as the Intel iWarp machines and MasPar machines allow the user to control routing. This provides an opportunity for the mapping software to design routings tailored for specific applications to minimize communication overhead.

The placement and the routing steps described above aim at minimization of communication overhead incurred by the topological mismatch. Different communication technologies of a multicomputer require use of different metrics for these steps. For a multicomputer, many switching techniques have been used and communication overhead varies from one scheme to the other. To achieve good performance, one needs to understand these techniques well. In the following, the communication techniques of store-forward, wormhole and other schemes such as circuit switching and virtual cut-through routing schemes are described.

Communication Switching Techniques

The last few years have witnessed rapid development in communication switching technologies of a multicomputer network. As opposed to the old store-forward scheme, new flow control schemes have been used and these schemes have reduced communication overhead significantly. Among them, the wormhole routing and its variation, the circuit switching scheme, are the most popular ones. For example, Intel's iPSC/2 and iPSC/860 adopt the circuit switching technique, and Intel's iWarp, Intel's Paragon, NCUBE's NCUBE/2 and Ametek's Symult 2010 use the wormhole routing scheme. In the following, we review store-forward and these three newer techniques.

In an old generation multicomputer, communication through an intermediate processor may interrupt local computation on that node (an example is Intel's iPSC/1). In a new generation multicomputer, a node usually consists of three components: a compute engine, a local memory module and a communication module (also called a router). The compute engine, coupled with the local memory, is responsible for local computation. The communication module, along with communication channels, is responsible for message passing with other nodes in the network. Intermediate message passing does not interrupt local computation.

Communication in a multicomputer network is determined by two methods: *flow control* and *routing*. Flow control is the method used to regulate traffic in a network. It determines when a message or part of a message can advance and what communication resources can be allocated to a message. Routing is a method to choose a path for a message over a network. Most commercial machines use a fixed routing scheme which always routes messages through fixed routes. In the following,

we will describe different flow control schemes and in the next section, we will describe routing schemes.

Three concepts are important to understand a flow control scheme.

- *Message:* The logical unit of communication between processes.
- *Packet:* The smallest unit which contains routing information. A message is divided into one or more packets.
- *Flit:* The smallest unit which is transmitted as a unit. A packet is further divided into one or more flits. Only the first flit contains the routing information.

We call the time to transmit a message from its source node to its destination node *message latency.*

In the following, we discuss message latency under an assumption that messages are never blocked during their entire transmission (the contention-free assumption). In the store-forward flow control scheme, a packet is treated as a single flit. In an intermediate node, a packet is received and buffered completely before it is forwarded to its next node. Let D be the number of hops a packet has to travel, and let L be the packet length. The message latency T_1 for a packet to reach its destination, under the contention-free assumption is

$$T_1 = D(w + L/b)$$

where w is a system dependent constant representing overhead for packet enqueuing and other bookkeeping, and b is the the bandwidth of a communication channel.

In the virtual cut-through or wormhole flow control scheme, a packet is divided into small flits. The first flit, also called the header, contains the routing information.

Communication channels are allocated flit by flit. The header reserves the communication channels it has traversed, which are allocated to its following flits. The last flit (also called the tail) releases a channel once it has passed across the channel. Since flits pass through channels in a pipelined fashion, the time for an entire packet to reach its destination, under the contention-free assumption is

$$T_2 = s + h * D + L/b$$

where h is the time to transmit the header across one channel and s is the startup time, i.e. the overhead for message injection into the network. In the current generation of multicomputers which utilize wormhole routing, h is small. For example, in iWarp, h is less than $20ns$ per byte [14].

Virtual cut-through and wormhole flow control schemes differ from each other when message contention is present. When the next channel for the header is unavailable, virtual cut-through buffers all the rest of the flits in the node where the header is. On the other hand, the wormhole flow control scheme is a blocking version of virtual cut-through in that all flits stay in the buffers where they are when the next channel for the header is unavailable. It can be seen that the wormhole flow control scheme requires substantially smaller buffer space. On the other hand, since flits stay in the network when a message is blocked in the wormhole flow control scheme, they may block other messages.

Another flow control scheme which has been used in commercial machines is called circuit-switching which is similar to the one used in a telephone network. In the circuit-switching scheme, when a packet is to be sent to its destination, its header, which contains routing information, is first transmitted along its intended path and

reserves the channels if it is possible. After the path is established, the whole packet is sent out. Message latency under the contention-free assumption is similar to that of wormhole routing.

Figure 3 illustrates the store-forward, wormhole and circuit-switching schemes.

It can be seen that when there is no contention, the wormhole scheme and the virtual cut-through scheme have considerably less overhead than the store-forward scheme. Furthermore, when $L >> D$, the distance (number of hops traveled) has negligible effect on the transmission time for the wormhole, virtual cut-through and circuit-switching schemes.

Because of such a dramatic message latency improvement from the store-forward to the wormhole routing, some industry vendors [26] claim that task placement is no longer an important issue. For example, in Intel's iPSC/2 user guide [26], it is claimed that *"Direct-Connect routing imposes virtually no added penalty on multiple node communication. Consequently, you can view the machine as an ensemble of fully interconnected processors"*.

The above formulae, however, are based on the contention-free assumption. When messages are congested due to communication resource (channel or buffer) contention, the extra time caused by the contention may vary dramatically for different flow control schemes. To be more precise, the latency (T) of a message is the summation of the message latency (t) under the contention-free assumption and the time (C_t) spent due to communication resource contention. From the above description of the flow control schemes, many factors may affect T. In the store-forward flow control scheme, t is sufficiently large and therefore minimizing the message distance, which is the dominant factor for t, can be chosen as the major criterion. In a virtual

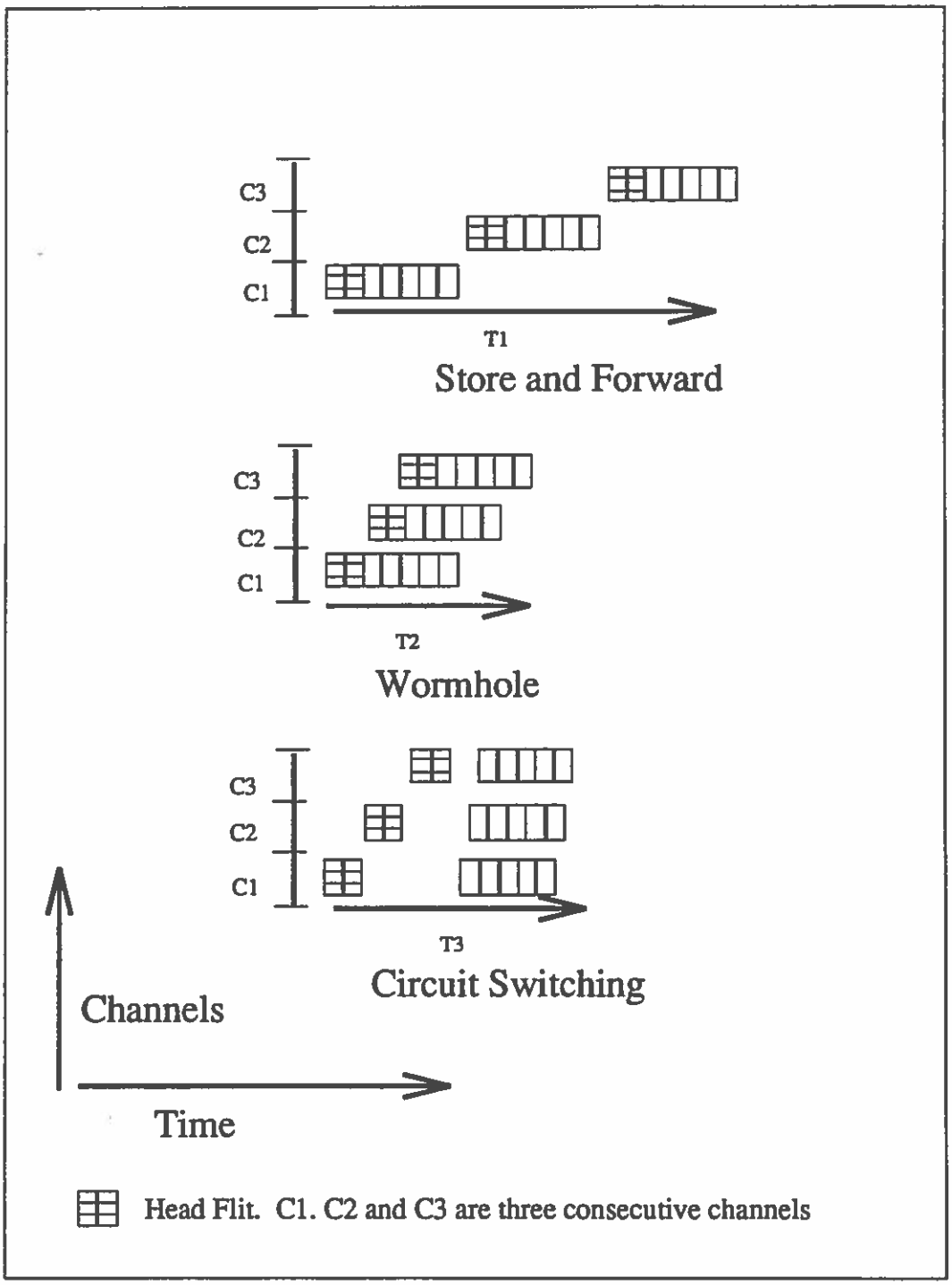


Figure 3: Illustration of Flow Control Schemes in a Contention-Free Situation.

cut-through, wormhole or circuit-switching scheme, however, t has dropped significantly and C_t's role in T has increased. We therefore need to re-examine metrics related to the mapping and routings.

Routings

Routing is the method used to choose a path for a message through the network. More precisely, a routing can be described as a permissible function $\mathcal{P}$ and a selection function $\mathcal{S}$.

$$\mathcal{P} : C \times N \mapsto 2^C$$

$$\mathcal{S} : \mathcal{P}(C) \times \beta \mapsto C$$

where C is the set of channels, 2^C is a power set of C, N is the set of addresses of nodes and β is a set of network states which may include current traffic state, past history of the routing or even random information (generated by a random generator). Function $\mathcal{P}$ identifies a set of *permissible* output channels in 2^C, given the current input channel $c \in C$ and the destination address $n \in N$. The selection function $\mathcal{S}$ chooses the output channel from the permissible channels $\mathcal{P}(C)$, based on the current network state β.

Depending on $\mathcal{P}$ and $\mathcal{S}$, we can classify routings into fixed (deterministic), oblivious and adaptive categories. A deterministic routing chooses a path based only on the source and destination addresses of a message (i.e., $\mathcal{P}$ returns a single-element set for the output channel and β is an empty set). In an oblivious routing, β does not contain any information on the current traffic state of the network but may contain some other information such as time, randomness or the contents of messages. Finally, an adaptive routing is the most general and it may make a routing decision based on the current traffic state.

Deterministic routing has been widely used in multicomputers because of its simplicity and good performance when the network traffic is light. For example, many hypercube multicomputers (iPSC/2, Ncube) use E-cube routing, which chooses the outgoing channel for a message as the first most significant bit where the destination address and the current node address differs. In the standard hypercube binary code labeling scheme, it can be seen that E-cube routing routes a message from higher dimension (more significant bit) to lower dimension. For example, if a message is to be sent from 0010 to 1001 in a four dimensional binary cube, the path chosen by the E-cube routing is $0010, 1010, 1000, 1001$. A similar routing scheme, called XY-routing, is used in a two dimensional mesh. The XY-routing always routes a message first along the X-direction (horizontal) and then along the Y-direction. The Intel Paragon and iWarp machines adopt this routing scheme as their default routing.

A classic oblivious routing scheme was proposed by Valiant and Brebner [116]. It first routes a message from its source to a randomly selected intermediate node and from there to its destination. In this routing, β contains some contents of the message (whether the message has reached its random intermediate node) and randomness information. It has been shown that for a permutation routing (i.e., a node sends only one message and receives only one message), this routing has $O(\log N)$ time on an N node cube. Another oblivious routing scheme allows the router to choose the outgoing channel based on the instruction given in the message header. In such a routing scheme, the user can control the routing by appropriately setting the routing information in the message header. This has been adopted in the iWarp system [49].

Recently, several adaptive routing strategies have been proposed and studied. In these adaptive routing schemes, randomization is used to diffuse traffic. Two

problems should be considered in designing an adaptive routing, namely, deadlock and livelock. Deadlock may occur because of circular waiting for channels (or buffers) (see, Section on page 26 for more detail). Livelock is a situation where a message moves indefinitely through the network and is never delivered to its destination. Ngai and Seitz [88] proposed an adaptive routing scheme for a virtual cut-through network. The routing uses randomness to misroute messages (i.e., send the message to a random output channel) once they are blocked and uses multiple buffers to solve the livelock problem. Another adaptive routing technique which uses randomness to solve the livelock problem has been proposed by Konstantinidou and Snyder [63]. The routing scheme is called the Chaos router because it uses random information to avoid the livelock problem with a high probability. Other adaptive routings for wormhole routed networks have been also proposed and studied [47, 46].

A restricted routing called permutation routing has been extensively studied in theory. Most work along this direction is based on the store-forward model. In a store-forward model, a permutation routing is closely related to the problem of sorting N elements on an N node network where initially, each node has one element. Many results, along with sorting results, are proposed for various topologies such as cubes, meshes and shuffle exchange networks [71]. Recently, there have been some interests in developing permutation routing on a wormhole or virtual cut-through routed network [83, 95].

Deadlock Avoidance

A routing may result in deadlock due to circular waiting for communication resources (buffers or channels). Deadlock can be avoided by either a careful routing or through additional hardware support.

Store-forward, virtual cut-through and wormhole routing schemes all can result in deadlock if messages are waiting for communication resources in a circular fashion. In store-forward and virtual cut-through, the communication resources are the buffers for messages to be relayed in a node. Figure 4 shows a possible situation if the buffer space in a node can only hold one packet.

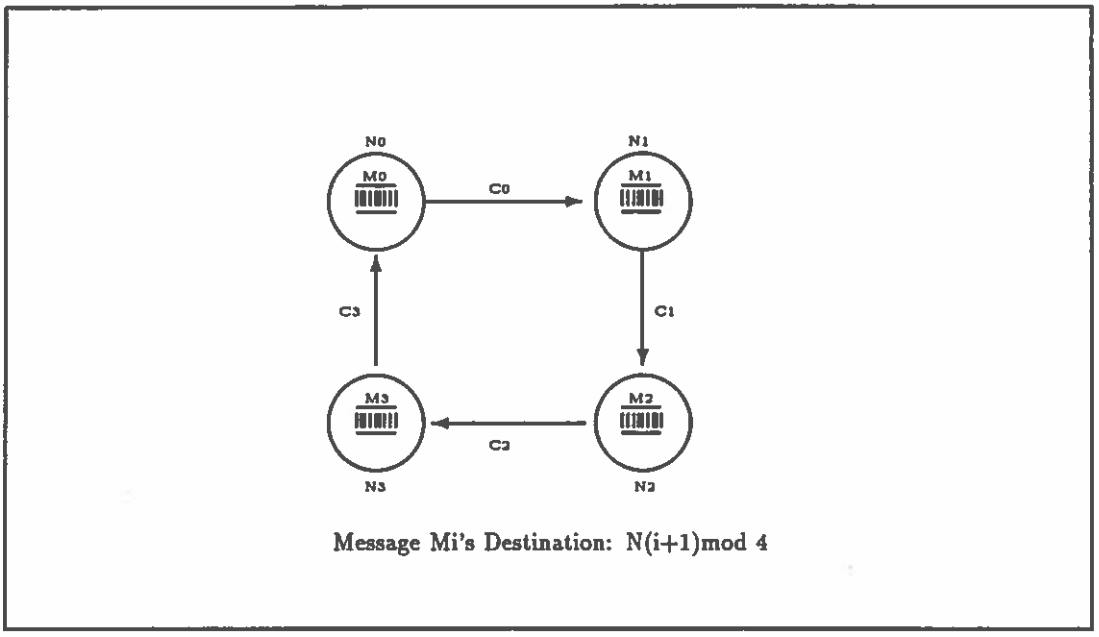


Figure 4: Deadlock in a Store-Forward Routing.

In wormhole routing, since the buffer space in a node for a relayed message can only hold one or two flits for each communication channel, it is more likely to result in a deadlock due to the occupancy of communication channels. Figure 5 shows such a possible situation.

More precisely, if there is a possibility for a routing to relay a packet or part of a packet from resource A to resource B, we say that resource A depends on B. This defines a binary relation $\mathcal{D}$ over all resources. If $\mathcal{D}$ is not partially ordered, then there is a possible circular dependency among resources, which corresponds to

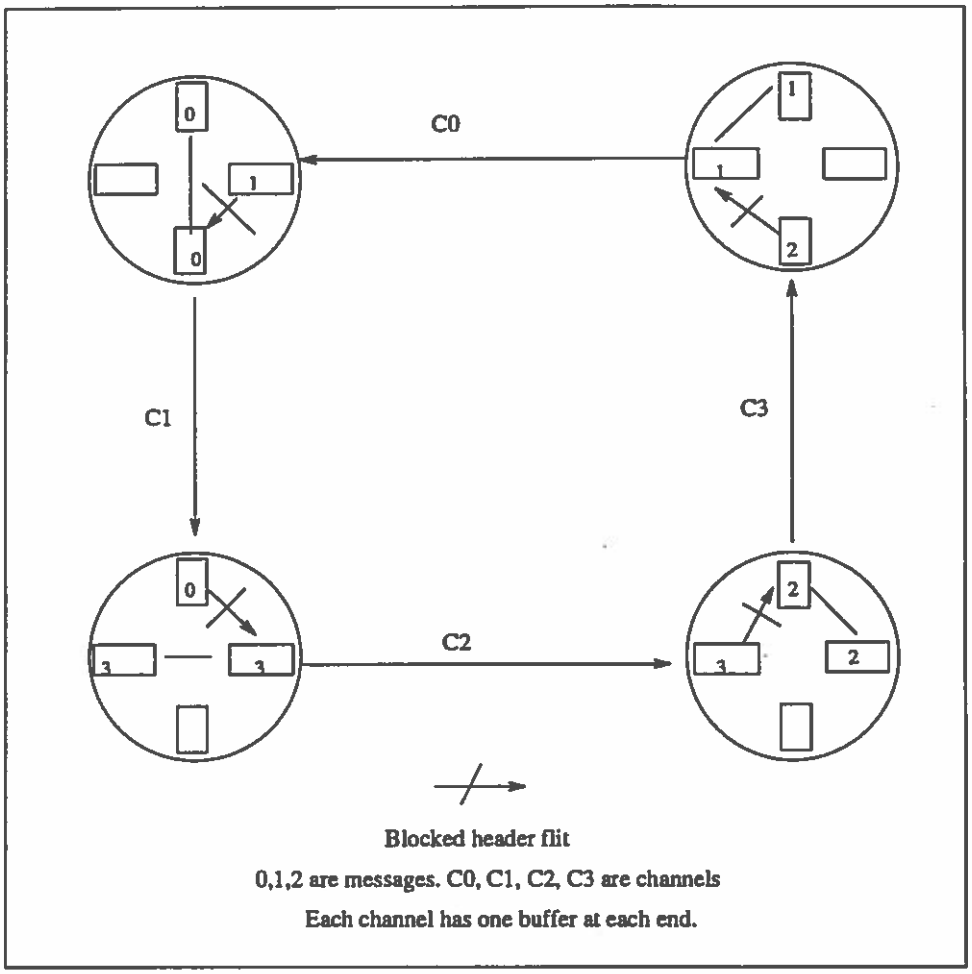


Figure 5: Deadlock in a Wormhole Routing.

a potential deadlock situation. In a wormhole routing, the resources are the channels and it has been shown that there exits no deadlock iff the channel dependency graph for a routing is acyclic [30].

Thus, to test whether a wormhole routing creates a potential deadlock situation, we can construct the channel dependency graph (CDG) based on the routing and then test whether the CDG has a cycle or not. In Chapter IV, we propose another graph called the Generic Physical Channel Dependency Graph (GPCDG) to capture all the possible channel dependencies as well as their physical connections in a single structure. Such a structure facilitates the development of low-contention deadlock-free wormhole routing.

Two approaches have been adopted to avoid a deadlock. The first is to carefully design a routing strategy such that the channel dependency graph is guaranteed to be acyclic. This approach minimizes the hardware support but may eliminate some possible paths, hence reducing the connectivity of the network. The other approach is to design flow control to avoid deadlock with appropriate hardware support. Extra buffers are introduced. We discuss these two approaches in the following.

A way to avoid deadlock is to simply adopt a deterministic (fixed) routing which can be proved to be deadlock free. For example, both E-cube routing for a hypercube system and XY-routing for a two-dimensional mesh have been proved to be deadlock free with respect to store-forward and wormhole routing schemes [30], respectively. This is because the channel dependency graph for these two routings are acyclic.

A fixed routing scheme, however, eliminates too many potential paths. This may result in poor utilization of channel resources. Although some schemes have been proposed [22, 46, 73] to allow finding more paths which are deadlock free, many

paths provided by the network with rich connectivity such as a hypercube are still not usable.

A general scheme to provide more connectivity while avoiding deadlock is to create virtual resources which share the same physical resource. The flow control is modified to guarantee freedom from deadlock. Analogous to methods used to avoid deadlock in an operating system [90], a partial order on such resources is introduced to ensure that a routing is designed so that the partial order is satisfied and no circular waiting state is reached.

In a store-forward routing, a method called the *structured buffer* [43] is used to label buffers in ascending order. Packets, once buffered, can only be sent to a restricted set of buffers on the next node. A buffer dependency graph is created so that if there is a packet which can be routed from buffer A to buffer B, then A depends on B. Again, the graph should be acyclic to ensure deadlock freedom. A simple implementation of the structured buffer scheme creates $D + 1$ buffers in every node where D is the network diameter. A packet can only be sent from a buffer labeled as n to another buffer whose label is larger than n. For example, the deadlock in Figure 4 can be avoided by introducing two more buffers for all nodes and the three buffers are labeled as $0, 1, 2$. When the packet is injected into the network, it is first buffered in buffer 0 and then it can be only sent to buffers which have larger labels. Figure 6 illustrates this idea. It can be seen that this scheme requires a considerable amount of storage space for a large network.

A similar technique called the *virtual channel* technique is used in wormhole routing to break any cyclic dependencies [30, 29]. The idea is to view communication channels as the resources for which different packet routes are competing. Multiple

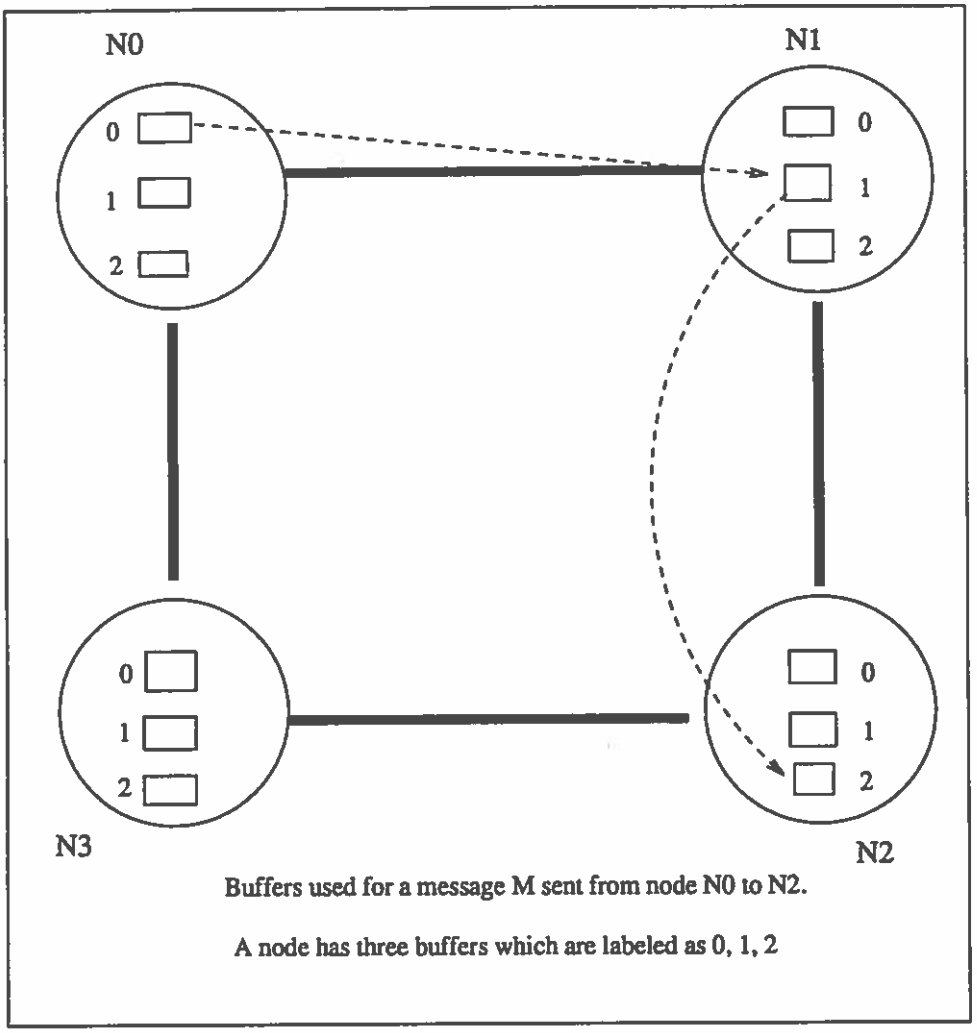


Figure 6: Deadlock Avoidance in a Store-Forward Routing.

flit buffers are introduced for a channel. These multiple flit buffers are called virtual channels since they are essentially responsible for communication over the same physical channels. Again, a labeling scheme is introduced to ensure the resource dependency to be acyclic. The deadlock in Figure 5 is avoided by introducing one more logical flit buffer c_0' and c_1' for each channel c_0 and c_1 respectively, as shown in Figure 7(a). The channel dependency graph is shown in Figure 7(b). In addition to its use for deadlock avoidance, the virtual channel technique can also be used to increase network connectivity and support adaptive routing.

With virtual channels, a physical network can be also partitioned into several virtual networks. A virtual network consists of a subset of virtual channels. Virtual networks make it possible to support multiple topologies on a single multicomputer. The concept has been used in some advanced computers such as Intel's iWarp system.

In [55], a simple way is proposed to partition an n-dimensional mesh into 2^n virtual networks such that all Manhattan shortest paths are represented. A Manhattan shortest path has a shortest rectilinear distance between the source and the destination. More precisely, suppose that G is a n dimensional mesh with nodes labeled with standard Cartesian coordinates. Clearly, a given channel lies in a single dimension. The direction of each channel in G has two possibilities, specified as 1 or -1. G is partitioned into 2^n virtual networks where each virtual network N is specified uniquely by a vector $(d_1, \ldots, d_n)$ where $d_i \in \{1, -1\}$. $N(d_1, \ldots, d_n)$ consists of all channels whose orientation is d_i in dimension i for each $i = 1, \ldots, n$. Figure 8 shows the partitioning on a 2-D mesh.

Remarks:

1. It is easy to show that the above partition covers all possible Manhattan shortest

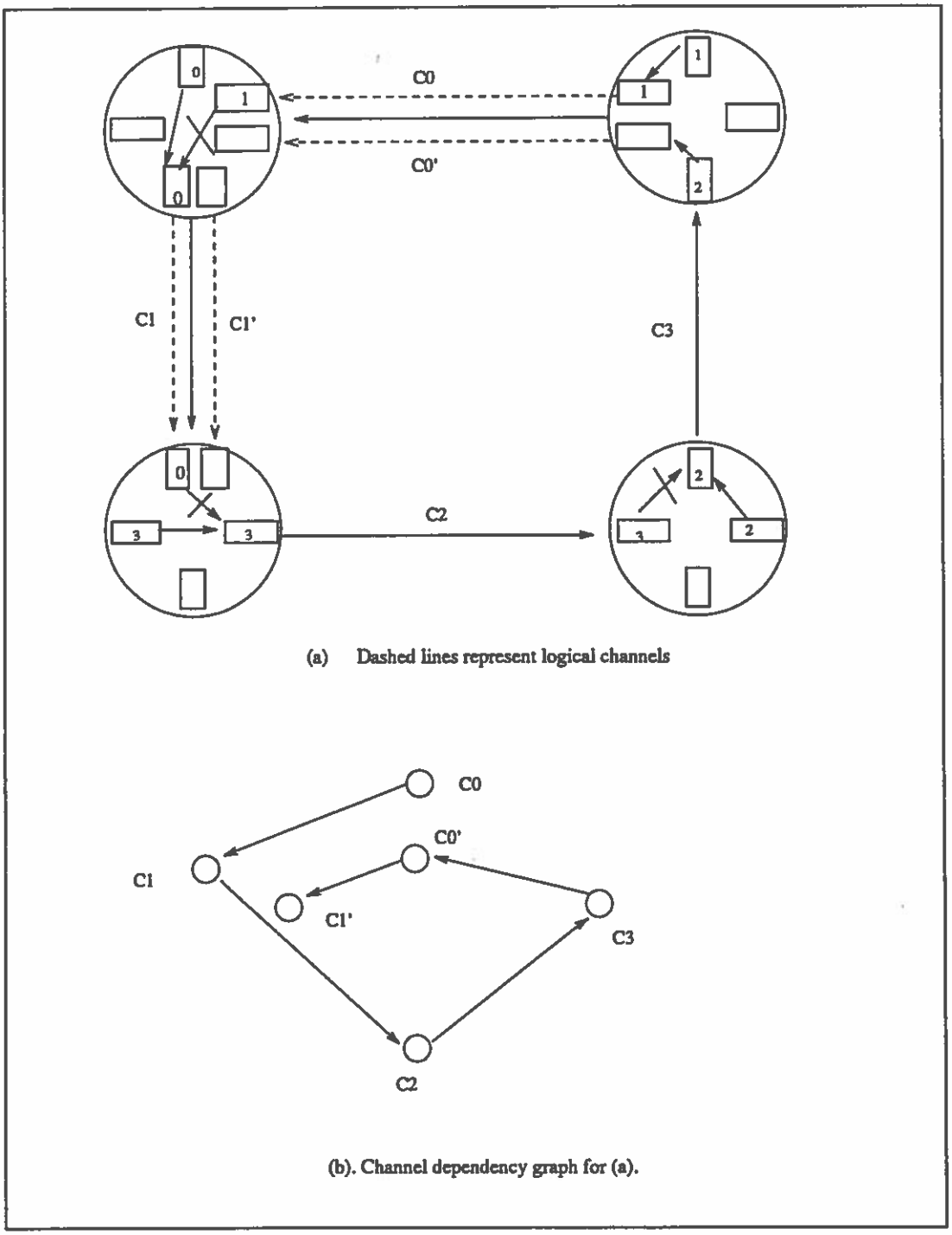


Figure 7: Deadlock Avoidance in a Wormhole Routing.

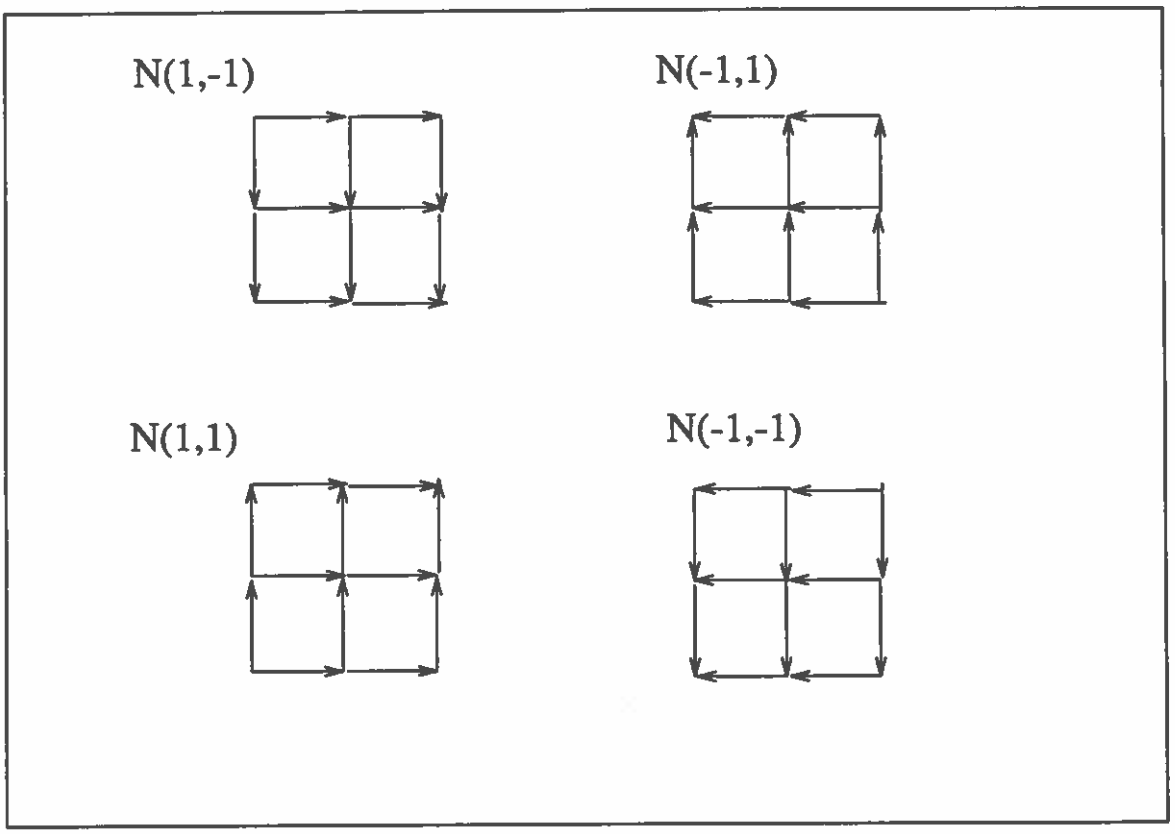


Figure 8: Partitioning of a 2-D Mesh into Four Virtual Networks.

paths in an n dimensional mesh.

2. If a routing routes message through a Manhattan shortest path in one of the virtual networks, then it is deadlock-free. This is because the virtual networks *themselves* are acyclic and hence their corresponding channel dependency graphs must also be acyclic.

The mesh which has such a virtual network support is called a *partitioned* mesh. The above scheme is best for a low dimensional mesh. For example, for a two dimensional mesh, the number of virtual networks is only two, which can be implemented practically. In fact, in the Intel iWarp system, two virtual networks are supported for a two dimensional torus, which is the physical configuration of the system [14, 49]. A similar virtual network partitioning scheme for k-ary n-cube has been proposed by Linder and Harden [74].

Fundamental Issues in Communication Overhead

In this section, we identify fundamental issues in developing mapping algorithms to reduce communication overhead on a multicomputer.

The store-forward routing scheme was used in old generation multicomputers such as Intel iPSC/1 and Caltech Cosmic Cube machines. This has been replaced with advanced communication technologies such as the wormhole and the circuit switching routing schemes in the current advanced multicomputers such as Intel Paragon, iWarp and Ncube Ncube-2 machines. As we can see from the previous section, message latency under the contention-free assumption has been reduced dramatically from the store-forward routing to the wormhole routing. This change has a great impact on the mapping problem. In the following, we identify three fundamental issues.

The first question is: *does task placement still matter for the performance of an application in an advanced multicomputer with the wormhole routing?* It is very natural to raise this question since the communication overhead has been reduced dramatically in the wormhole routing scheme. However, when the channel contention factor is taken into account, intuitively, message latency will be affected by traffic congestion. It is thus important to understand how much a task placement scheme can do to reduce message latency. Several researchers [13, 23] have studied this problem based on various limited assumptions. However, the total completion time of actual benchmarks which are mapped with several mapping schemes has not been directly measured and studied.

The second question is: *what factors contribute to the communication overhead?* For the store-forward routing, maximum dilation is one of the major metrics used to develop a mapping. This has been changed for the wormhole routing. It is therefore necessary to develop and study new metrics to better characterize the communication overhead.

The third question is: *how to reduce communication overhead when designing a mapping algorithm?* How do we design good placement algorithms and good routing algorithms to optimize the new metrics identified? New methods should be studied to incorporate these new metrics into the design of mapping algorithms.

In this thesis, we focus on these problems in Part I.

Communication Overhead Metrics

In this section, several mapping metrics related to communication overhead are described. For the purpose of this thesis, we ignore other mapping metrics which are not related to the communication overhead caused by the topological mismatch (such

as processor load balancing).

Since we do not consider processor resource mismatch in this thesis, we simplify the mapping and assume that contraction has been done and thus, only a one-to-one placement in the mapping is considered.

We first define the metrics based on the static task graph model. A static task graph is a weighted graph $C = (V, E, W_v, W_e)$ where the nodes V represent processes and edge $e = (a, b) \in E$ between node a and b represents a communication (possibly bi-directional) between task a and task b with weight $W_e(e)$. This weight represents an estimation of the communication volume between a and b. Each process a is further associated with a weight $W_v(a)$ which represents an estimation of the amount of computation performed by a. The topology of a multicomputer is modeled as a graph $A = (V(A), E(A))$, where $V(A)$ represent processors and $E(A)$ correspond to the processor-to-processor physical connections of the underlying interconnection network.

A mapping $\mathcal{M}$ can be specified by two functions: $\mathcal{M}_p$ and $\mathcal{M}_r$ where $\mathcal{M}_p$ is a function which maps nodes in V to nodes in $V(A)$. $\mathcal{M}_p$ is called a *placement* function. $\mathcal{M}_r$ is a function which maps an edge (a, b) in E to a path $p_0, p_1, \ldots, p_m$ in $E(A)$ such that $p_0 = \mathcal{M}_p(a), p_m = \mathcal{M}_p(b)$. $\mathcal{M}_r$ is called the *routing* function. When the architecture only provides a fixed routing, $\mathcal{M}_r$ is completely determined by the fixed routing function.

The most widely used metrics for inter-processor communication overhead include *dilation*, *channel contention* and *path level contention*. Based on the static task graph model, we have the following definitions:

Definition 2.1

The *dilation* of an edge $m \in E$ is $|\mathcal{M}_r(m)|$ (denoted as $Dila(m)$). The *weighted dilation* of m is $W_e(m) \times |\mathcal{M}_r(m)|$ (denoted as $WDila(m)$). □

The *dilation* represents the the number of hops a message has to traverse.

Definition 2.2

The *channel contention* of a channel c in $E(A)$ is defined as $|\{e \in E \mid c \in \mathcal{M}_r(e)\}|$ (denoted as $Cont(c)$). The *weighted channel contention* of c is defined as

$$\sum_{\{m \in E | c \in \mathcal{M}_r(m)\}} W_e(m)$$

(denoted as $WCont(c)$). □

The channel contention measures how many messages conflict on a single channel. Recently, a new metric called *path level contention* which is more useful for the wormhole or the circuit-switching flow control schemes, is proposed by Chittor [23]. Intuitively, *path level contention* is defined as, for each communication edge $e \in E$, the number of other messages whose paths intersect with the path for e.

Definition 2.3

The *path level contention* of a communication edge $m \in E$ is defined as $|\{m' \in E \mid \mathcal{M}_r(m') \cap \mathcal{M}_r(m) \neq \emptyset\}|$ (denoted as $PLC(m)$). The *weighted path level contention* of a communication edge $m \in E$ is defined as

$$\sum_{m' \in E | \mathcal{M}_r(m') \cap \mathcal{M}_r(m) \neq \emptyset} W_e(m')$$

(denoted as $WPLC(m)$). □

The above definitions can be generalized to the TCG model. Intuitively, a TCG can be viewed as a sequence of communication phases which are represented as a static task graph. Thus, we can use the metrics defined above for each phase. Furthermore, we can define metrics for the overall TCG based on the metrics for each phase.

Traditionally, maximum dilation has been the major metric used to evaluate a placement scheme. However, as we can see from the previous sections, this is no longer correct for advanced multicomputers. Maximum channel contention and path level contention have become more important [23].

The above metrics are important since they give us an easy way to measure a mapping with respect to communication overhead. We will show how these communication overhead metrics affect the ultimate metric, which is the total completion time of a mapped program. In the next chapter, we will empirically study these metrics based on simulation.

Overview of Part I

The studies in Part I concentrate on the fundamental issues proposed on page 35. The contributions of Part I are summarized as follows.

In Chapter III, we carry out empirical studies for two typical benchmarks through simulation. The simulation results of these two benchmarks indicate that task placement may still significantly affect the performance of a mapping. These results also indicate that the performance of a mapping can be affected by many factors including traffic congestion, message size, message startup cost and machine size. One can not simply rely on a single factor to characterize the mapping performance. To better quantitatively characterize and evaluate communication overhead, we validate message latency formulae proposed in [76] through simulation. Fur-

thermore, a method is proposed to incorporate these formulae into an event-driven multicomputer simulator, which leads to a useful performance evaluation framework for parallel programs.

Chapter IV and Chapter V develop application-specific routings to reduce communication overhead. We show that carefully designed routing algorithms that are sensitive to the communication structure in an application are effective as a key step in mapping for high performance. In Chapter IV, a general framework for application-specific routing is proposed. An efficient heuristic is developed to generate low communication overhead deadlock-free routing for a general multicomputer network. In Chapter V, we propose an efficient heuristic to generate application-specific routing on a mesh-connected multicomputer with virtual channel support.

CHAPTER III

COMMUNICATION OVERHEAD ON A MULTICOMPUTER

The purpose of this chapter is to better understand the nature of communication overhead and how it affects the mapping problem for the wormhole routing, one of the new communication technologies First, we carry out empirical case studies for two well known benchmarks which are mapped with several mapping schemes. The mapping schemes are carefully chosen to clearly distinguish the contribution of individual mapping metrics to the total completion time of the benchmarks. By directly measuring the total completion time of a benchmark through simulation, we show that the extra communication overhead incurred by the topological mismatch can significantly affect program performance. Based on our simulation results, the impact on the performance of the benchmarks of several factors including path-level contention, dilation, message startup cost, and system size is also characterized. Furthermore, we develop analytical message latency formulae and show that they accurately predict the actual latency. We then propose a method to incorporate the formulae into a general-purpose multicomputer simulator, which leads to an important parallel program performance evaluation framework.

Related Work

In the past, there have been several studies on communication overhead in a multicomputer network. We summarize these studies in the following.

In [28], Dally proposed an analytical model for the performance of a k-ary n-cube and showed that, as long as the message injection rate is constant, the network traffic contention will increase as the average message path length increases. Since the larger the number of nodes in a system, the longer the average message path length will be, this result justifies the fact that communication contention will have more serious effects when the system size is large. This study was directed to the network performance of wormhole routing. The method used in this study is probabilistic. No specific benchmarks were studied.

Agarwal [1] proposed a contention model for a buffered direct network. Based on the assumption that the probability of a network request (i.e. a message sending request) on any given cycle from a node processor is constant, a message latency formula was derived:

$$T_c = h + \frac{B(1 + n\rho)}{1 - \rho}$$

where T_c is the average message latency for a message sent from one node to another node h hops away, B is the number of flits for a message, n is the dimension of the mesh, and ρ is the probability that a given channel is occupied.

The formula is more applicable to statistical performance studies, since the probability of the message sending request for all processor nodes is assumed to be a constant. In an application, the message request rate may vary dramatically from process to process and thus from processor to processor too. Thus, it is unclear how accurate the formula will be if one approximates the message sending request rate as a constant.

Chittor's work [23] directly addressed the effects of communication contention related to the mapping problem. In his study, he assumed that a task injects (sends out) messages uniformly to its neighbors at a constant rate (the message injection rate). He studied the effects of a random mapping of a task graph on a mesh-connected multicomputer. He showed that the message saturation injection rate is inversely proportional to the maximum *path-level contention*, which is the maximum number of messages whose paths intersect with the path of a given message. This result implies that the more contention a mapping yields, the lower the peak message injection rate for a task can be, which in turn, limits the attainable speedup. Furthermore, an artificial application whose task graph is the same as matrix transpose was performed on the Symult 2010 and the results conform with the above analytical results. As in Agarwal's work, no message latency formulae directly related to an application were developed. Since the message saturation injection rate was used as the main performance criterion, the total completion time of an application was not directly studied. The other limitation of Chittor's work is that only random mapping was compared with a specific mapping. Since a random mapping may have several bad factors such as path-level contention and dilation, it is hard to distinguish among the contribution of individual metrics to the mapping effect.

Bokhari [12] performed several simple experiments on Intel's iPSC/860 which uses the circuit-switching flow control scheme. He studied the effects of message size, distance, communication channel contention, and nodal contention on message latency. Using simple artificial communication structures, he concluded that edge contention leads to severe overhead for all message sizes. A similar approach was adopted by Dunigan [34] to study the performance of Intel's iPSC and NUBE ma-

chines.

The major drawbacks in the above approaches are:

- the effect of different mappings on the total completion time of real parallel applications was not studied;
- the factors which may affect the mapping performance were not clearly distinguished;
- the message latency formulae developed are not sensitive to an application and are not directly applicable to performance prediction for an application.

In this chapter, we carry out empirical case studies on two well-known benchmarks to directly measure the mapping effect on the total completion time. The results offer a qualitative explanation for the mapping effect. We then proceed to study the problem of quantitatively predicting message latency. The message latency formulae are further validated through simulation.

Multicomputer Network Simulation

Our empirical study is based on a multicomputer network simulator, Proteus [16]. The Proteus simulator was originally developed at MIT. The simulator is a direct-execution, event-driven simulator. It supports the concurrent C language with extensions for interrupted-based message passing and shared memory support such as semaphores. The simulator was implemented on the Pmax, a DEC Station. Processes are modeled as user-level threads, which reduces context switching overhead dramatically. A library of non-local computations such as process-spawning and interrupt-based message passing primitives are provided. A program is first compiled

into MIPS machine code and then its basic blocks are identified and augmented with cycle-counting operations (for simulation time advance). Cycles spent in the basic block are estimated based on a table which maps a MIPS instruction to the number of cycles needed for the instruction execution. The augmented program is directly executed. The simulator is event-driven with all the events being put into a central event queue. Proteus provides an accurate simulation for both processors and networks [17].

We customized the network simulation modules of the Proteus simulator. A PICL-like (Portable Instrumented Communication Library [42]) library was implemented on Proteus so that the customized simulator is capable of supporting a general concurrent C programming language with standard message passing mechanisms. Furthermore, both wormhole and store-forward flow control schemes were implemented. Appendix A shows a program for 2-d FFT using PICL-like communication primitives.

In the following, we first describe the network simulation modules that we developed for the wormhole control flow scheme. We then describe how to modify the simulation modules to model the store-forward flow control scheme. In the simulator, a message is always assumed to be a single packet, that is, we do not distinguish a message from a packet.

There are two types of network events: SEND_PACKET and ROUTE_PACKET. SEND_PACKET is the initial network event generated when a message send operation is invoked. A ROUTE_PACKET event is generated for intermediate hop routing. In general, to route a message n hops away, if the message is not blocked, one SEND_PACKET and $n-1$ ROUTE_PACKET events are generated. The last

ROUTE_PACKET event delivers the message to the destination node processor.

In the blocking case, the blocked event is enqueued into a waiting FIFO queue associated with the busy channel. When the channel is released by the current message, the first blocked event is put back into the central event queue.

The Simulation Algorithm

The algorithms to handle the two events are described as follows. The central event queue is denoted as CEQ. The time for passing a single flit through a channel is modeled with a constant *Flit_Time*.

- *SEND_PACKET event handler:* A SEND_PACKET event is generated when a message send function in the program is called. The SEND_PACKET event handler mainly does two things: 1) get the route for the message; 2) try to inject the message into the network (if it is successful, a ROUTE_PACKET event is generated). The message startup cost is modeled with a variable *Startup_Cost*. Figure 9 describes the handler in more detail.

- *ROUTE_PACKET event handler:* A ROUTE_PACKET event is generated to model the event of passing the message header across an intermediate channel. The ROUTE_PACKET event handler first checks whether or not the event is resumed from the suspended queue. If it is, the handler can not simply advance its tail right away since there might be several events which are demanding the same channel for the header. If such a race condition occurs, the event which is processed first wins and will occupy the channel for its header. Other events which demands the same channel will be suspended again. Such a policy corresponds to a FIFO. Figure 10 describes the handler in pseudo code.

```
1. if    (Source == Destination)
         dispatch the message to Destination processor;
   else  call ROUTER to obtain channel C and next node N;
   endif;
2. if    (C is free)
         mark the channel BUSY;
         Timestamp = current Timestamp + Startup_Cost +
                     Flit_Time;
         ADVANCE_TAIL of the message;
         enqueue a ROUTE_PACKET event for N into CEQ, FINISHED;
   else  enqueue a SEND_PACKET event
         into the associated suspend queue of C, FINISHED;
   endif;
```

Figure 9: SEND_PACKET Event Handler.

```
1. if    (the event is not resumed from the suspended queue)
         ADVANCE_TAIL of the message;
   endif;
2. if    (Current == Destination)
         if    (Tail == Destination)
               dispatch the message to Destination processor;
         else  enqueue a ROUTE_PACKET to CEQ, FINISHED;
         endif;
   else  call ROUTER to obtain channel C and next node N;
   endif;
3. if    (C is free)
         mark the channel BUSY;
         Timestamp = current Timestamp + Flit_Time;
         enqueue a ROUTE_PACKET event for N to CEQ, FINISHED;
   else  enqueue a ROUTE_PACKET event
         into the associated suspend queue of C, FINISHED;
   endif;
```

Figure 10: ROUTE_PACKET Event Handler.

- *ADVANCE_TAIL:* The routine advances the flits in the tail position and if there are no flits left for the tail processor, the channel from the tail to the next processor is released. Notice only for the source processor, it is possible that it has more than one flit. Once the tail flit leaves the source processor, a tail processor will have only one flit (i.e., the tail flit). A ROUTE_PACKET event in the waiting FIFO queue associated with the released channel is put back into the central event queue CEQ with the Timestamp updated to the current time. Notice also since the intermediate channels which are occupied by flits other than the tail and the header do not change their state for such a movement, it is sufficient to only handle the tail and the header.

- *ROUTER:* The network topology and the routing decision are simulated in this routine. It is straightforward to implement the E-cube and XY routing routines.

A Simple Example

In the following, we describe a simple example for the wormhole flow control scheme. Suppose that simultaneously, node 1 sends a message with 3 flits to node 3 and node 2 sends a message with 2 flits to node 3. The router chooses 1-2-3 as the route for the first message and chooses 2-3 as the route for the second message. The central event queue is denoted as CEQ. An event is represented as a triple

(*type (Source-Destination), timestamp, current-node*)

where *type* is either SEND_PACKET or ROUTE_PACKET, *(Source-Destination)* is used to denote a message whose source and destination nodes are *Source* and *Destination* respectively, *timestamp* is the timestamp for the event, and *current-node* denotes the current node where the message header of the event is.

In this example, we assume Startup_Cost to be zero cycles and the cost to pass a flit across a channel to be one cycle. Figure 11 shows every state when an event in the central event queue (CEQ) is removed and handled, and Figure 12 illustrates the states pictorially.

```
1. CEQ: {(SEND_PACKET (1-3), 1, 1), (SEND_PACKET (2-3), 1, 2)}
   state: all channels are free
2. CEQ: {(SEND_PACKET(2-3), 1, 2), (ROUTE_PACKET (1-3), 2, 2)}
   state: 1-2 busy
3. CEQ: {(ROUTE_PACKET (1-3), 2, 2), (ROUTE_PACKET (2-3), 2, 2)}
   state: 1-2 busy, 2-3 busy
4. CEQ: {(ROUTE_PACKET (2-3), 2, 3)}
   state: 1-2 busy, 2-3 busy with (ROUTE_PACKET (1-3), 2, 2) suspended
5. CEQ: {(ROUTE_PACKET (2-3), 3, 3)}
   state: 1-2 busy, 2-3 busy with (ROUTE_PACKET (1-3), 2, 2) suspended
6. CEQ: {(ROUTE_PACKET (1-3), 3, 2)}
   state: 1-2 busy, 2-3 free
7. CEQ: {(ROUTE_PACKET (1-3), 4, 3)}
   state: 1-2 busy, 2-3 busy
8. CEQ: {(ROUTE_PACKET (1-3), 5, 3)}
   state: 1-2 free, 2-3 busy
9. CEQ: {} time: 6
   state: 1-2 free, 2-3 free.
```

Figure 11: Simulation Steps to Pass a Message from Node 1 to Node 3 and to Pass Another Message from Node 2 to Node 3.

The simulation of store-forward flow control can be easily achieved by modifying the above modules. In fact, if the flit size is treated as the whole message size instead of a fixed small number of bytes, the flow control becomes store-forward.

The Mapping Effect

As we point out on page 41, past studies in characterizing communication overhead have a major drawback, namely, the total completion time of an actual parallel

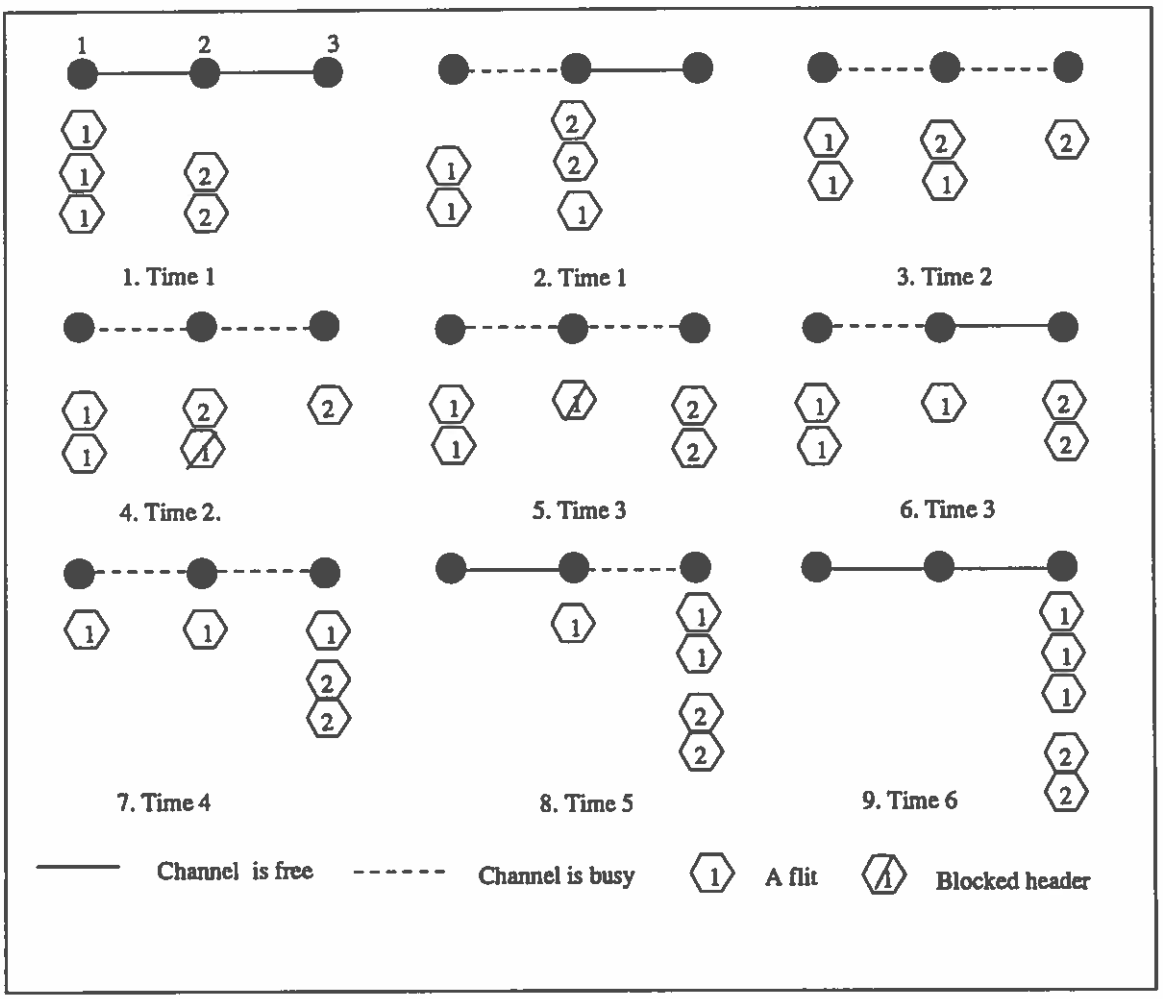


Figure 12: The State of Each Channel and Physical Time of Each Step to Pass a Message from Node 1 to Node 3 and to Pass Another Message from Node 2 to Node 3.

program is not measured. Furthermore, individual metrics' contribution to the mapping effect is not clearly identified. To understand the relationship between an actual program and its performance regarding communication overhead, we adopt a different approach which uses two well-known benchmarks. Distinct mapping schemes clearly demonstrate individual metrics' contribution to the mapping effect.

The Benchmarks

Two benchmarks are carefully chosen for the study. The first one is a divide and conquer algorithm (called DAQ) which is modeled as a binomial tree [76]. The second one is a 2-D FFT which is modeled as a butterfly structure [40]. The criteria we used to choose the benchmarks are: 1) Benchmarks should have wide applications. Our first benchmark represents an important paradigm in parallel programming, divide and conquer, which applies to applications ranging from sorting to multiplication of a series of matrices. The second benchmark FFT has wide applications in numerical analysis and digital signal processing. 2) Benchmarks should have interesting communication structures for which various mapping schemes will have clearly differing performance impacts. For a binomial tree structure, several mapping schemes have been developed for both store-forward and wormhole routed systems [76]. The relatively rich interconnection structure of the butterfly used in 2-D FFT causes different mappings to have distinct communication overheads due to contention as well as dilation. 3) Benchmarks should have regular temporal behavior so that we can analyze their behavior more precisely. Many parallel applications exhibit a *logical* synchronous phase-by-phase temporal structure. For example, for a typical divide and conquer algorithm which is modeled as a complete binary tree, the computation proceeds in a phase-by-phase fashion, that is, at every phase, all nodes except for the

leaf nodes at the same level of the tree do local computation (i.e. dividing) and then send messages to their two children. In this case, we can view that a phase consists of the local computation, the computation phase, and the message sending carried out at a level of the tree, the communication phase. The phase-by-phase model has been elaborated and studied in the TCG model [78]. Both DAQ and FFT have phase-by-phase communication structures and the communication intensity varies from phase to phase.

A DAQ is modeled as an N-node binomial tree. A typical divide and conquer algorithm consists of $\log(N)$ dividing phases where a node repeatedly receives message from its parent node, does computation, and then passes messages down to its children; and $\log(N)$ combining phases where a node receives messages from its children, does computation, and then passes the results up to its parent node. Since the dividing stage is similar to the combining stage, in our actual benchmark run, we only model the dividing stage. Figure 13 shows an 8-node binomial tree and the three dividing phases. Message sizes in our DAQ benchmark are assumed to be uniform in the whole computation. Uniform message sizes for the DAQ occurs in applications such as the multiplication of a series of matrices where the resultant matrix is passed up to parent nodes from children nodes. The local computation of the DAQ application is modeled as a loop of dummy integer additions. The number of iterations of the loop is equal to the size of the message. Appendix A shows a sample of the program.

For a 2-D FFT, we use a butterfly model. The algorithm is first described in [40]. In such an algorithm, the first $\log(P)$ phases (P is the number of processors) are the communication and then combination phases, while the last phase involves

a sequential FFT on each node. Figure 14 shows the static topology (which is a hypercube) for a 3-phase FFT and its phases.

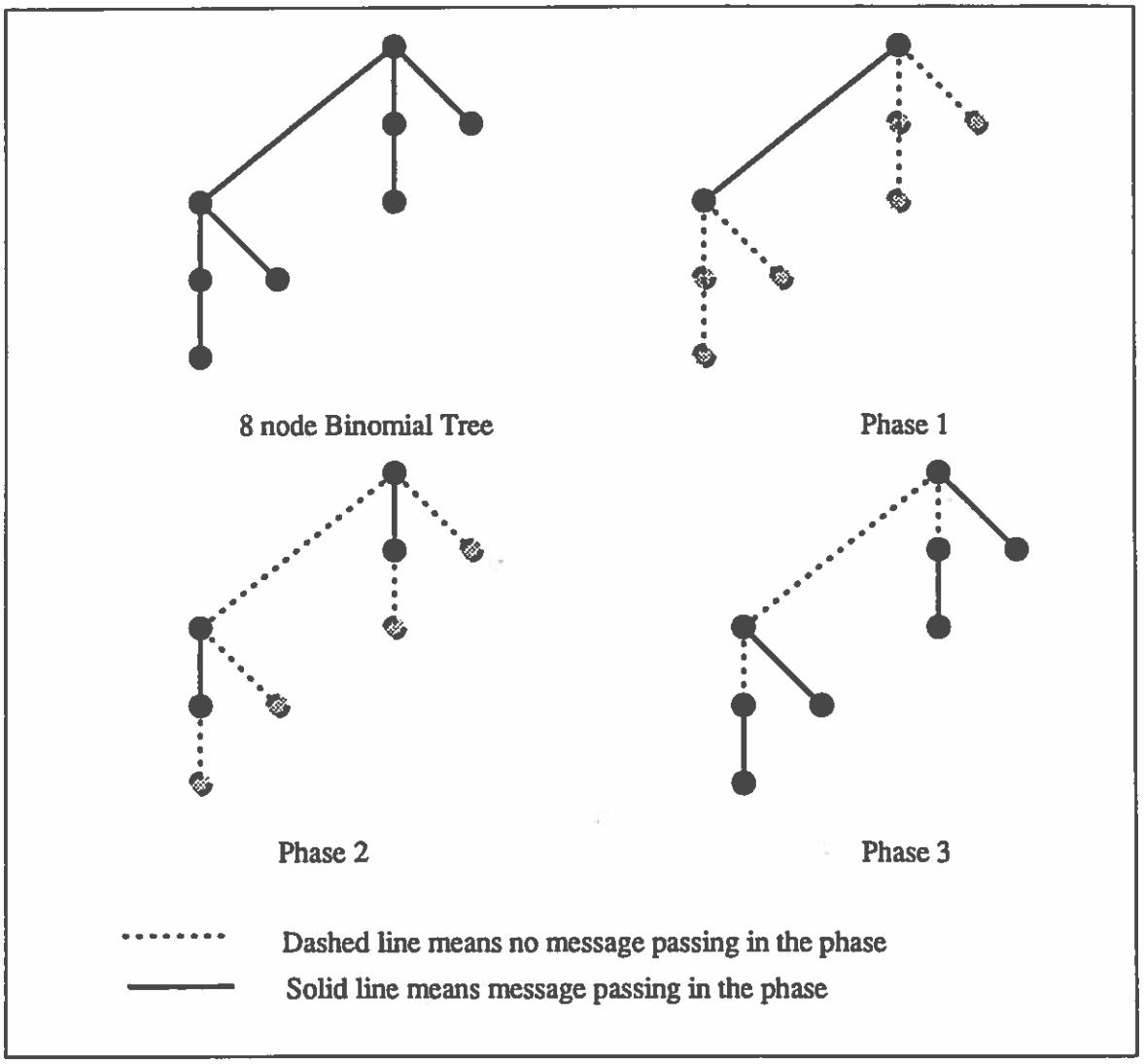


Figure 13: An 8-Node Binomial Tree and Its Three Phases.

Based on the metrics defined on page 36 in Chapter II, in the following, we generalize the metrics with respect to a TCG model which is used here to describe the two phase-by-phase benchmarks. Formally, a TCG for a phase-by-phase benchmark

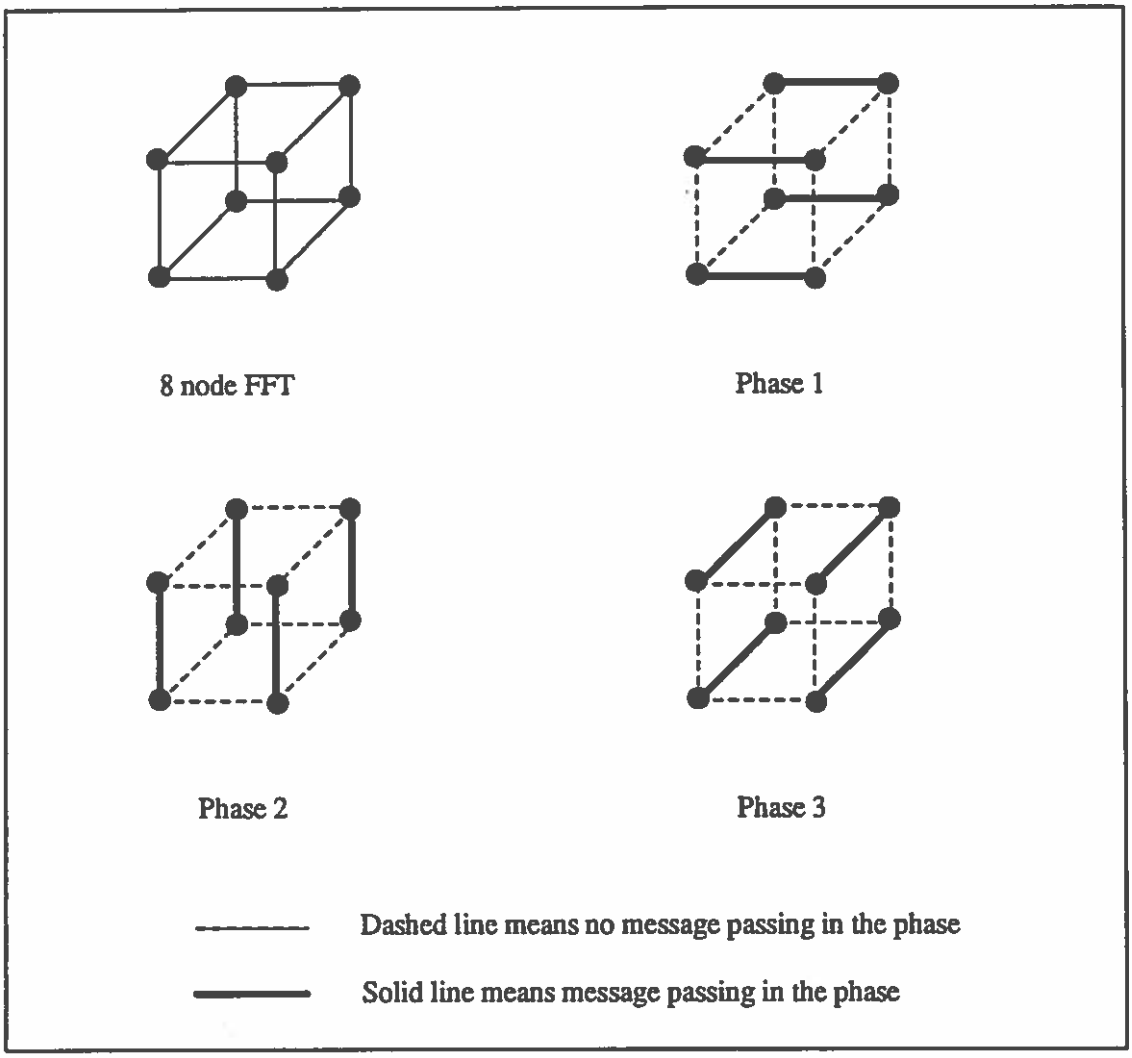


Figure 14: An 8-Node FFT Topology and Its Three Phases.

consists of K phases: each phase i is denoted as a static graph P_i (as we can see from the examples in Figures 13 and 14).

Definition 3.1

Path-level contention:

- For each phase i, $mplc(i)$ is the maximum path-level contention for graph P_i (see the definition on page 36).
- The Maximum phase Path Level Contention (denoted as MaxPLC) is defined as the $\max_{i=1}^{K}(mplc(i))$.
- The Total Maximum phase Path Level Contention (denoted as TotMaxPLC) is defined as $\sum_{i=1}^{K} mplc(i)$.

□

Definition 3.2

Dilation:

- For each phase i, $mdila(i)$ is the maximum dilation for graph P_i. Furthermore, $totdila(i)$ is the total dilation for graph P_i (see the definition on page 36).
- The Maximum phase Dilation (denoted as MaxDila) is defined as $\max_{i=1}^{K}(mdila(i))$.
- The Total Maximum phase dilation (denoted as TotMaxDila) is defined as $\max_{i=1}^{K} mdila(i)$.
- The total phase dilation (denoted as TotDila) is defined as $\sum_{i=1}^{K} totdila(i)$.

□

Definition 3.3

Contention:

- For each phase i, $mcont(i)$ is the maximum contention for graph P_i.
- The Maximum phase Contention (denoted as MaxCont) is defined as the $\max_{i=1}^{K}(mcont(i))$.

□

The above metrics can be used to predict the total completion time for a phase-by-phase application. Intuitively, if the path level contention is the dominant factor for mapping performance, TotMaxPLC serves as an approximation to the total completion time. The same is true for TotMaxDila if dilation is considered as the dominant factor for mapping performance.

For a DAQ, we consider three different mappings: reflecting, growing and random. The first were are developed and formally defined in [76]. Figure 15 shows an example of the reflecting mapping. Intuitively, the reflecting mapping is constructed recursively as follows: for the base case, we construct the mapping to map a one-node binomial tree to a one-node mesh. To map a DAQ to a mesh of size $2^m \times 2^{(m+1)}$, we divide the mesh consists of two submeshes of size $2^m \times 2^m$ and use the mappings constructed for mapping two DAQ subtrees of size $2^m \times 2^m$ to the two submeshes. We then reflect one submesh horizontally. Finally, we place the two submeshes horizontally and designate one of the roots of the submeshes as the root of the original DAQ. In the case where a mesh is of size $2^m \times 2^m$, we can construct the mapping similarly from the mapping to a mesh of size $2^{(m-1)} \times 2^m$.

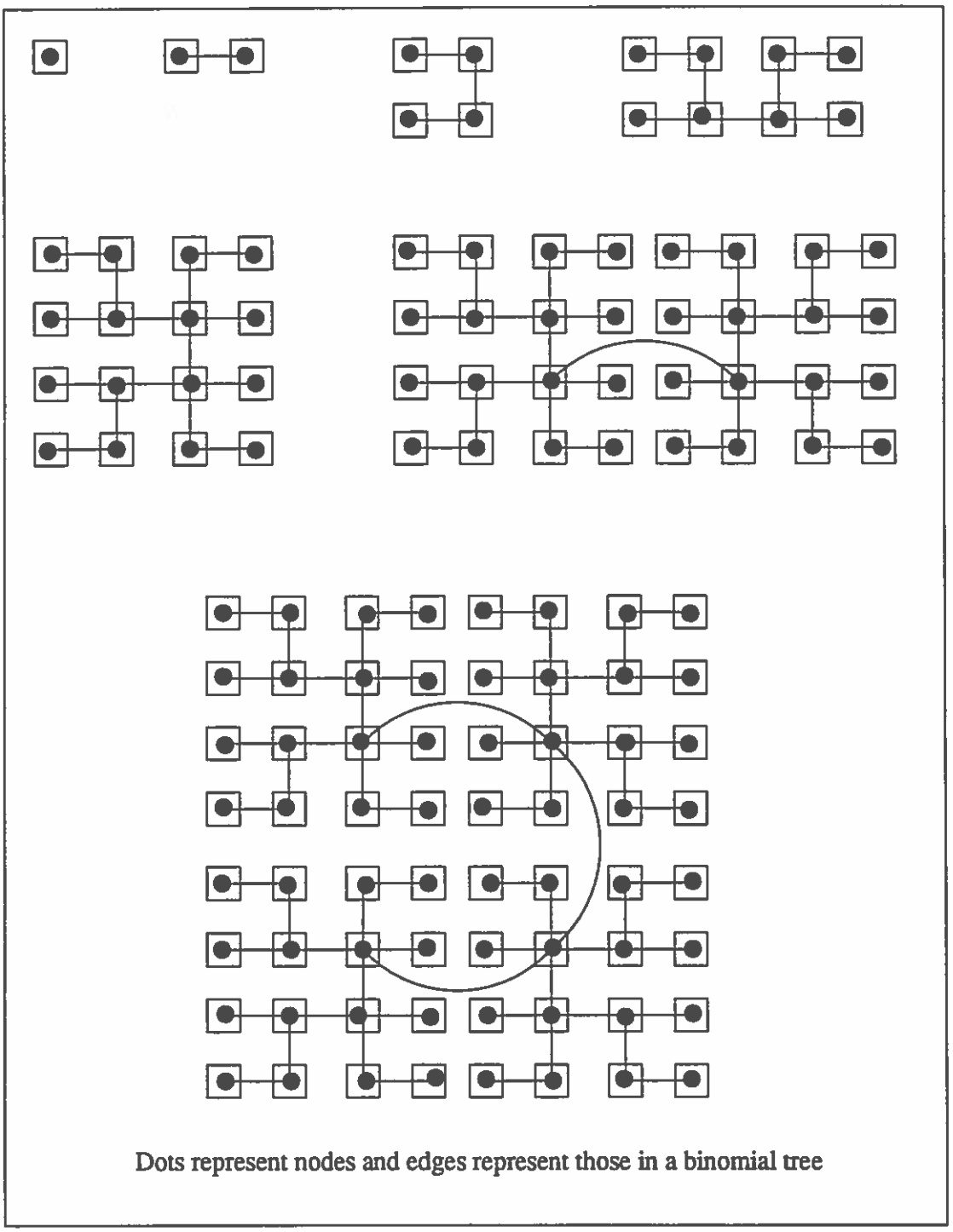


Figure 15: The Reflecting Mapping to Meshes with Size from 1 to 64.

The second mapping called the growing mapping is also constructed recursively. Figure 16 shows an example of the growing mapping. Again, we start from the trivial mapping for a one-node binomial tree. To construct the growing mapping to a mesh of size $2^m \times 2^{(m+1)}$, we consider the case where the mesh consists of a center submesh of size $2^m \times 2^m$ and two other submeshes of size $2^m \times 2^{(m-1)}$, placed on the left side and right side of the center submesh respectively. Since a binomial tree of size $2^m \times 2^{(m+1)}$ can be considered as "growing" an additional node from each node of a binomial tree of size $2^m \times 2^m$, we can place those "grown" nodes on the two side submeshes so that they are in the same row as their parents and they are of the distance from their parents for all the "grown" nodes.

The random mapping is constructed by using a uniform random generator to assign a process to a randomly generated processor.

Table 1 shows MaxPLC, MaxCont and TotMaxPLC for the two mappings. Table 2 shows the TotDila, MaxDila and TotMaxDila for the two mappings. From the tables, we observe that for the contention metrics, the reflecting mapping has no path-level contention at all (i.e. equal to one, the minimum value), and the growing mapping has path-level contention that increases logarithmically with mesh size. For the dilation metrics, the reflecting mapping always has larger TotMaxDila than the growing mapping (see Table 2). Such a distinction between the two mappings allows us to differentiate the effects of contention and dilation on the performance. Analytical results from [76] for these two mappings state that the reflecting mapping is good for a wormhole-routed network while the growing mapping is good for a store-forward network. Our work here also provides an empirical verification for these claims.

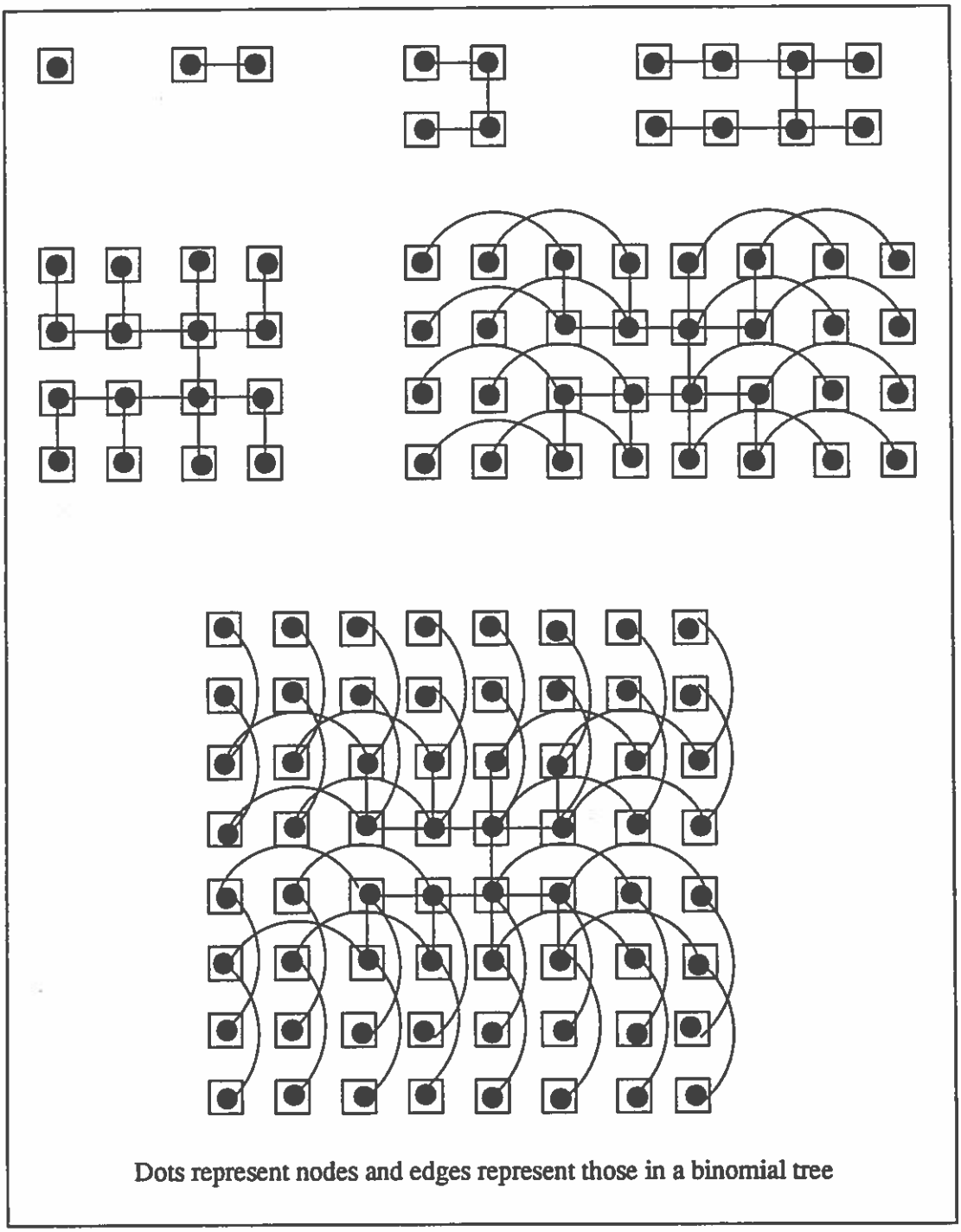


Figure 16: The Growing Mapping to Meshes with Size from 1 to 64.

Table 1: Contention Metrics for the Reflecting and Growing Mappings of DAQ

Mesh Size	Reflecting			Growing		
	MaxPLC	TotMaxPLC	MaxCont	MaxPLC	TotMaxPLC	MaxCont
64	1	6	1	2	8	2
256	1	8	1	4	18	4
1024	1	10	1	8	34	8

Table 2: Dilation Metrics the Reflecting and Growing Mappings of DAQ

Mesh Size	Reflecting			Growing		
	TotDila	MaxDila	TotMaxDila	TotDila	MaxDila	TotMaxDila
64	69	3	10	111	2	8
256	291	5	20	879	4	16
1024	1197	11	42	7023	8	32

For the FFT benchmark, three different mappings are considered: gray-code, identical and random. The gray code mapping scheme is first introduced in [40]. It can be described as follows: first, all nodes in an FFT are labeled with the gray code scheme so that a node only communicates with others whose node labels differ in one row-bit. This results in a hypercube structure for the static task graph of the FFT. We then map the hypercube to the mesh with a row-column gray code mapping scheme. Figure 17 and Figure 18 show the example mappings for gray-code and identical mappings. Table 3 and Table 4 show the contention and dilation metrics of the gray code and identical mappings. From the tables, we can see that while the gray code mapping has smaller TotMaxPLC on all the meshes, the identical mapping has smaller TotMaxDila.

Table 3: Contention Metrics for the Gray Code and Identical Mappings of FFT

Mesh Size	Gray code			Identical		
	MaxPLC	TotMaxPLC	MaxCont	MaxPLC	TotMaxPLC	MaxCont
64	4	14	4	6	18	4
256	18	50	8	20	58	8
1024	73	196	16	72	201	16

Table 4: Dilation Metrics for the Gray code and Identical Mappings of FFT

Mesh Size	Gray code			Identical		
	TotDila	MaxDila	TotMaxDila	TotDila	MaxDila	TotMaxDila
64	896	7	22	896	4	14
256	7680	15	52	7680	8	30
1024	63488	31	115	63488	16	60

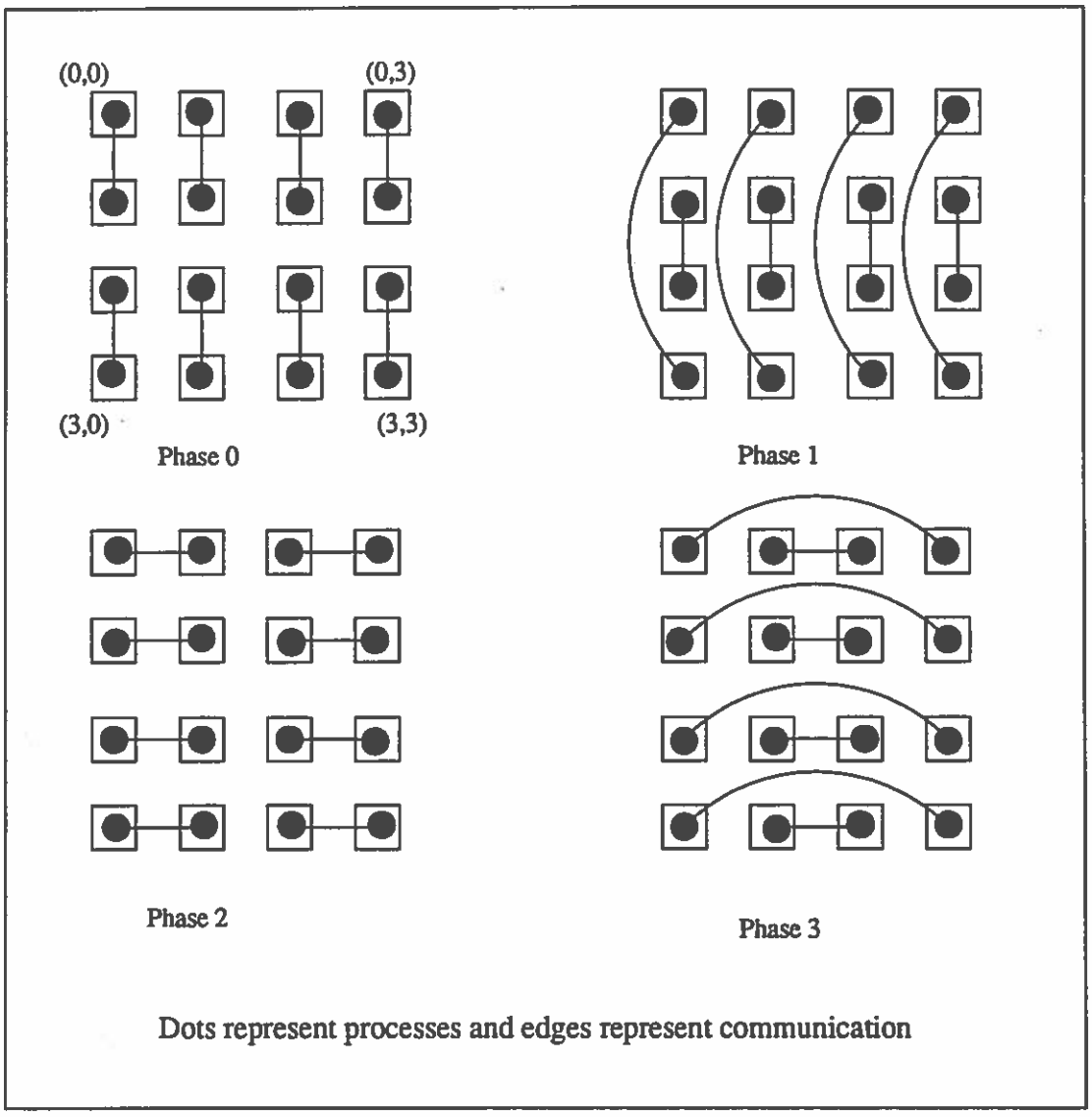


Figure 17: The Gray Code Mapping (Shown by Phase by Phase Communication) to Mesh of Size 16.

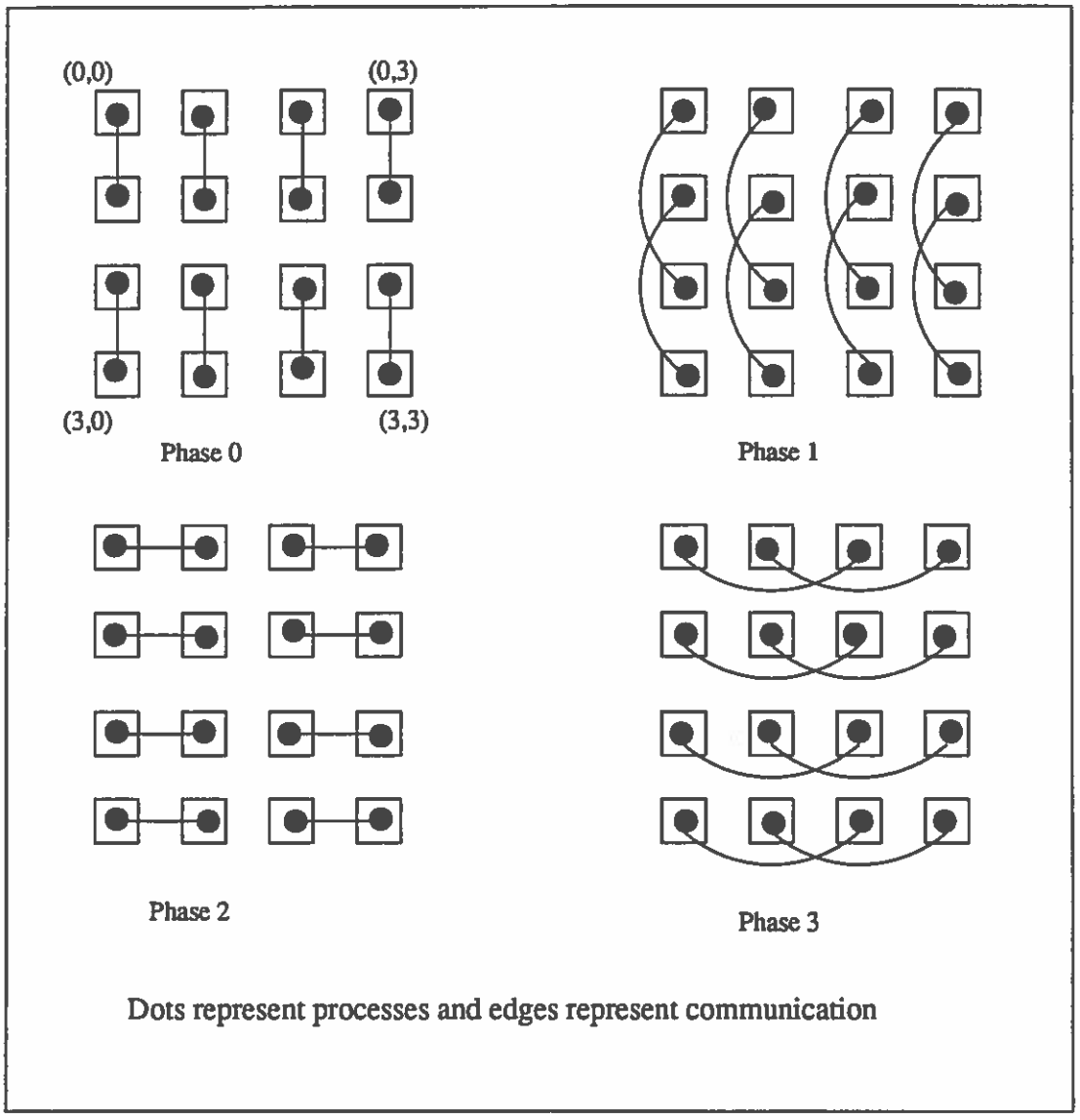


Figure 18: The Identical Mapping (Shown by Phase by Phase Communication) to Mesh of Size 16.

Simulation Setup

The architecture topology we study is a 2-D mesh. This is because the mesh topology is becoming more dominant in the latest commercial machines due to its good performance [28]. A simple XY-routing is used for the mesh architecture.

The empirical studies are organized to study the performance of different mapping schemes with various problem sizes, namely, in DAQ, the message size parameter (8, 128, and 8192 bytes) and in FFT, the size of the subarrays processed in the first phase (8, 64, 512 and 4096 bytes). Furthermore, we consider the performance impact by the following architectural parameters, namely, flow control schemes (wormhole and store-forward), system size (i.e. the number of processors in a mesh, 64, 256 and 1024 nodes are considered) and the startup cost. The message startup cost is chosen to be a fixed number of cycles, 100 simulation cycles. The message size parameter is varied relative to the fixed startup cost. In this way, the effect of the startup cost can be characterized based on its relation to message size. For example, for a message with only 8 bytes, the startup cost with 100 cycles will play a more significant role with respect to the message latency than for a message with 4k bytes.

The simulator is configured so that passing one byte across a channel takes one simulation cycle. In the wormhole control flow scheme, the flit length is assumed to be one byte. Thus, passing a flit across a channel under wormhole routing takes one simulation cycle while passing a message of L bytes across a channel under store-forward takes L simulation cycles.

In all the simulation runs, the completion time of the benchmark run (i.e. the number of simulation cycles produced at the end of the simulation run, called simulated time *stime*) is observed and recorded. By comparing the completion time of

benchmarks of the same problem size mapped with different schemes, we are able to observe the performance differences between two different mappings.

Simulation Results

The complete simulation results are listed in Appendix A. Based on the simulation data, we make the following observations.

- *Message traffic congestion:* Message traffic collisions can significantly affect the performance of a mapping for large message size on a wormhole-routed system. For example, for the DAQ benchmark, the reflecting mapping has no traffic collisions and the growing mapping has some traffic collisions. The performance difference between these two mappings is clearly demonstrated in Table 5. In Table 5, for a mesh of 1024 nodes, the total completion time of the growing mapping is two times larger than that of the reflecting mapping.

  The message traffic congestion also affects mapping performance for the FFT benchmark. The performance difference between the gray code and identical mappings is shown in Table 6. It is interesting to note that in the FFT case, the performance difference between the gray code mapping and the identical mapping is not as large as in the DAQ case. Furthermore, the simulated completion time (stime) of the gray code mapping is slightly worse than that of identical mapping on a 64 node system. The reason for this is that since the gray code mapping has worse TotMaxDila (see Table 4), performance is also affected by this factor.

  When message size is small, path-level contention is no longer a dominant factor. For example, for the cases where message size is 2 bytes or 128 bytes, the growing

mapping performs slightly better than the reflecting mapping on all the mesh sizes (refer to Tables 13 to 15 in Appendix A). This result implies that one can not simply single out one factor to characterize the mapping effect.

Table 5: Performance Comparison Between the Reflecting Mapping and the Growing Mapping of DAQ of Message Size Equal to 8192 Bytes on a Wormhole-Routed System. Column 4 Shows the Ratio of stime of the Growing Mapping over stime of the Reflecting Mapping

Mesh Size	stime(Reflecting)	stime(Growing)	Ratio
1024	99017	206402	2.08
256	72810	119397	1.64
64	53119	68282	1.29

Table 6: Performance Comparison Between the Gray Code Mapping and the Identical Mapping of FFT of Message Size Equal to 512 Bytes on a Wormhole-Routed System. Column 4 Shows the Ratio of the stime of the Identical Mapping over the stime of the Gray Code Mapping

Mesh Size	stime(Gray)	stime(Identical)	Ratio
1024	998167	1282705	1.28
256	876964	942890	1.07
64	814084	813135	1.0

- *Message dilation:* Message dilation is a more important factor for a store-forward routed network. For example, in DAQ, the total maximum phase dilation of the reflecting mapping is larger than that of the growing mapping. For message size equal to 128 bytes, Table 7 shows that the growing mapping consistently performs better than the reflecting mapping in a store-forward routed system. Message dilation can also affect performance for small message sizes on a wormhole routed system. For example, in the DAQ benchmark, the growing

mapping *consistently* performs slightly better than the reflecting mapping on all the mesh sizes (refer to Tables 13 to 15 in Appendix A).

Table 7: Performance Comparison Between the Reflecting Mapping and the Growing Mapping of DAQ of Message Size Equal to 128 Bytes on a Store-Forward Routed System

Mesh Size	STime(Growing)	STime(Reflecting)	Ratio
1024	19676	22935	1.17
256	8650	9734	1.13
64	4574	5030	1.10

- *System size:* The simulation results confirm the conclusion drawn in [1, 23] that the bigger the system size is, the more important a mapping is. This can be seen from Tables 5,6, and 7.

- *Message startup cost vs. message size:* The relationship between message startup cost and message size plays an important role for performance. In our simulation, the mapping effect is more visible when message size well exceeds the startup cost. In Table 13, for example, the performance difference between the reflecting mapping and the growing mapping is too small to be noticed when message size is only 2 or 128 bytes, but the difference becomes obvious when message size becomes larger, 8192 bytes. The reason here is that since the startup cost is modeled as 100 cycles, for size equal to 2 or 128 bytes, the startup cost offsets the message latency and thus different mappings do not make much difference.

- *Random mapping:* Since no knowledge about the computation structure is taken into account, a random mapping should have worse contention as well as dila-

tion. In all of our cases, random mapping *consistently* performs worse than a carefully designed mapping (refer to Tables 13 to 21 in Appendix A).

From the simulation results, we can see that TotMaxPLC and TotMaxDila better predict the total completion time difference than other metrics such as TotDila and MaxCont. This is because TotMaxPLC and TotMaxDila better capture the temporal behavior.

To summarize, in this section, we have shown, from empirical case studies, that a mapping which is designed carefully to minimize message traffic collision and dilation can significantly outperform other mappings for both wormhole and store-forward flow control schemes. Our studies also show that the performance of a mapping can not be simply characterized by a single factor. In a wormhole-routed machine, for applications with large message size, path-level contention can be used as the main optimization criterion. For applications with small message size, however, both path-level contention and dilation should be taken into consideration to design a good mapping. Furthermore, our results strongly suggest that certain kinds of temporal information should be incorporated into mapping metrics. For example, in the FFT benchmark, the gray code mapping has a better (smaller) MaxPLC on 64-node and 256-node systems but has a larger MaxPLC on a 1024-node system than the identical mapping. However, the gray code mapping always outperforms the identical mapping in the large message size case. This is because MaxPLC ignores the accumulated path-level contention among all the phases in the FFT. On the other hand, TotMaxPLC takes the accumulated contention effect among all the phases into account and thus predicts the performance of a mapping more accurately (in the FFT case, the TotMaxPLC of the gray code mapping is always smaller than that of the identical

mapping).

In the next section, we will discuss how to model runtime information more accurately to better predict message latency and hence communication overhead.

Message Latency Estimation

From the above empirical results, we can see that different placements still have a great impact on the performance of a program. However, those results only qualitatively identify the possible factors which may affect performance. Furthermore, several factors such as message size and message startup cost can interact with each other to produce different performance results. Thus, to accurately predict the performance of a mapping, it is not sufficient to use simple static factors such as path-level contention or dilation from a static task graph, which do not take the runtime behavior (such as phase-by-phase information) into account. To better evaluate and characterize the performance of a mapping, we should develop a more accurate model for the runtime behavior of the mapped program. One approach to accomplish this is to develop a sufficiently accurate analytical formula to predict message latency. Based on the message latency formula, we can further develop analytical models to predict the total completion time of a mapped program. In this section, we study message latency prediction based on program, mapping and architectural parameters.

In [76], a message latency estimation formula was proposed to analyze mapping performance. The message latency formula assumes that we are given a communication structure where all messages are sent out simultaneously. For a message m, in a wormhole-routed machine, we will use the following formula, which is a variation of

the one in [76].

$$T(m) = startup + \beta * (\frac{plc(m)}{FLIT\_LENGTH} + dilation(m))$$

where, *startup* is the number of cycles for a message passing startup cost, β is the number of cycles needed to send a flit across a communication channel, *dilation* is the number of hops the message has to travel (in XY-routing scheme), and $plc(m)$ is the path-level contention of message m when m is sent. In order to estimate the total communication time, we assume that the message m will meet and be delayed by *every* message m_1, whose route is overlapped with m's. We further assume that the delay induced by each m_1 is equal to the time that m_1 needs to occupy a single link.

The first assumption is pessimistic, since there may not necessarily be a conflict merely because two messages happen to use the same link (the first message may arrive early enough that by the time the second message arrives, the link may be free). To understand the justification for the second assumption, it is obviously true if the messages contend exactly once for a single link. One of them is then delayed by precisely the time that the other one occupies the link. If two messages contend for a contiguous sequence of links, then they are sent in a pipelined fashion, one after the other. In this case, the delay for the first link is just what we have assumed above. For each successive link, the delay depends on the difference between the message volumes (if the message that is sent first is no larger than the later one, then there is no further slowdown). Finally, if two messages contend multiple times in disjoint sections of their paths our assumption is not valid, although it is often the

case that the first conflict will separate the messages enough that later contention is unlikely in any but the most pathological cases. In fact, for many fixed routing schemes such as the XY-routing on a mesh and the E-cube routing on a hypercube, it is guaranteed that the routes of two messages only intersect with each other on at most one continuous section.

For the store-forward flow control, the above formula can be easily modified as follows. The $FLIT\_LENGTH$ is equal to the message length $length(m)$, however, β is changed to $FLIT\_LENGTH * \gamma$ where γ is the number of cycles needed to pass a flit across a communication channel. The following is the formula for the store-forward scheme:

$$T(m) = startup + FLIT\_LENGTH * \gamma * (\frac{plc(m)}{length(m)} + dilation(m))$$

An impediment to the above two formulae is that they require knowledge of path level contention $plc(m)$, i.e., runtime information about how many messages have path level contention with m at the time m is sent. This limitation can be overcome in three ways. 1) When we are doing analytical studies of a known program which has regular temporal communication structures, the path-level contention of each message can be approximated. For example, in the DAQ benchmark, at each phase, we know what messages are sent and also, based on the mapping, we can calculate path-level contention of each message at each phase. In fact, the formulae were originally developed for this purpose. 2) In a simulator, we can calculate the path-level contention on the fly based on the network state when the message is sent. We will detail this scheme on page 73. 3) We can make a gross approximation based on

the static task graph if it is known. This approximation could potentially introduce a large error into the formulae.

Justifying the Formulae

In this section, we validate the proposed formulae with randomly generated communication structures, where each node can have at most one message sender and one message receiver. Three independent random generators are used, the first is for the message sender, the second for the message receiver, and the third for the message volume, which is bound to a given size.

The simulator is configured such that the startup cost is 100 cycles. The topology of the simulator is a 2-D mesh with 64 to 1024 nodes. The routing scheme used in the simulator is XY-routing. Three different message sizes are used, 8 bytes, 4k bytes and 8k bytes. Two flow control schemes are used: wormhole and store-forward. Each run is repeated 100 times to achieve a low standard deviation for the error measured.

Tables 8 and 9 list the simulation results. In the tables, we validate the proposed formulae by comparing the predicted simulated completion time (*pstime*) when the formulae are used in the simulator, with the simulated completion time (*stime*) when a detailed (hop-by-hop) simulation in the simulator is used. In the tables, the Avg. Error Ratio stands for the average of $(pstime - stime)/stime$ over 100 runs.

From the simulation results, we can see that predicted simulation time (*pstime*) matches that of the accurate simulation (*stime*) fairly well for both wormhole and store-forward routing. For the wormhole flow control scheme, about 14% average error ratio is achieved. The worst 28% average error ratio happens for small messages (8 bytes) on a large system (1024 nodes). The reason is that for small messages, the $plc(m)$ factor in the formula may be too conservative since when message size is small,

Table 8: Prediction vs. Simulation Error for the Wormhole Routing

Machine size	Msg Size (<= bytes)	Avg. Error Ratio	std_deviation
64	8	0.08	0.03
64	4096	0.15	0.14
64	8192	0.14	0.13
256	8	0.17	0.05
256	4096	0.13	0.12
256	8192	0.12	0.08
1024	8	0.28	0.06
1024	4096	0.08	0.07
1024	8192	0.10	0.07

Table 9: Prediction vs. Simulation Error for Store-Forward Routing

Machine Size	Msg Size (<= bytes)	Avg. Error Ratio	std_deviation
64	8	0.08	0.04
64	4096	0.26	0.11
64	8192	0.25	0.12
256	8	0.11	0.04
256	4096	0.21	0.09
256	8192	0.21	0.09
1024	8	0.16	0.06
1024	4096	0.20	0.07
1024	8192	0.20	0.06

messages whose path intersects with message m's may have quickly passed through some of the channels (or even their whole path) so that these messages may not have a conflicting demand for the same channel anymore.

For the store-forward flow control scheme, the average error ratio is increased to about 19%. This is again introduced by the $plc(m)$ factor which is more conservative in predicting the delay for a message due to the contention than in the wormhole scheme. The reason here is that, in the wormhole scheme, since a message can occupy multiple channels at the same time, it is more likely for two messages to compete for the same channel than in the store-forward case, where a message can only occupy a single channel at any time.

In the following section, we show how to incorporate the formulae into a performance evaluation framework using a simulator.

Incorporating the Message Latency Formulae into A Simulator

There are many applications of the message latency formulae. In this section, we show how to use the formulae for performance evaluation of a parallel program by incorporating the formulae into a simulator.

To use the formulae, we need to know the communication structure to calculate the path-level contention of a message. To be more accurate, we use the runtime communication structure constructed dynamically by the simulator when message sending primitives are executed. More precisely, when a message is sent, the communication structure is formed based on the current pending messages (called *pending message graph*) in the simulator. A message is called *pending* if the message is already injected to the network (i.e. its SEND_PACKET event has been processed) but is still not delivered to its final destination. These pending messages are recorded in

the simulator dynamically and when a message finishes its transmission, it is deleted from the pending message set. Based on the *pending message graph*, path-level contention can be calculated and the message formulae are used to calculate the message arrival time. A new ROUTE_PACKET event is generated with the calculated message arrival time as its timestamp. Notice we can use more flexible formulae here to predict the message latency. For example, for a message whose path is long, we can estimate the message latency from the source to the middle node (a middle node is the node where the path length from source to itself is approximately equal to the path length from itself to the destination) and the message latency from the middle node to the destination. This will generate two ROUTE_PACKET events instead of one and will give us a more accurate estimation if this is necessary. The extreme case here is that we will estimate the message latency hop by hop, which will result in the most accurate simulation but longer simulation time.

We show the use of the formula by running 2-D FFT on both simulators for the wormhole control flow scheme, one of which is based on the detailed, hop-by-hop network simulation and the other is based on the estimation formula for the network simulation. Table 10 shows the simulation results.

Compared with the error ratio introduced for a random communication structure on page 71, much smaller error ratios are achieved here. The reason for this is as follows. Since there is local computation performed in the FFT, the total completion time consists of two parts, namely, the time spent on the local computation and the time spent on message passing. Since the message passing time is only a certain percentage of the total completion time, the error ratio is reduced.

Table 10: Prediction vs. Simulation Error for a 2-D FFT on a 64 Node Wormhole Routed System

Subarray size	Mappings	Detailed Time	Prediction Time	Error Ratio
8	Identical	24315	23389	0.038
8	Gray	24012	23454	0.023
8	Random	23766	23276	0.021
64	Identical	180846	176078	0.027
64	Gray	181512	176241	0.029
64	Random	182913	176682	0.034
512	Identical	1815960	1792797	0.013
512	Gray	1821085	1793155	0.015
512	Random	1857309	1793378	0.034

Applications of the Message Latency Formulae

The message latency formulae proposed above has many interesting applications.

1. From the formulae, we can clearly state that path-level contention, dilation, as well as the startup cost, may all contribute to communication overhead introduced by a placement scheme in a wormhole routed system. Development of a good task mapping scheme should be tailored by these factors.

2. We can use the formulae to do analytical studies of the performance of a mapped program. This is demonstrated in the paper by Lo et al. which analyzes the performance of the reflecting mapping and the growing mapping for a binomial tree divide and conquer algorithm [76].

In the following, we detail the third application. We propose a practical scheme to integrate the techniques proposed above with the synthetic benchmark framework proposed by Poplawski [92] for parallel program performance prediction.

A synthetic benchmark for an application program is constructed as follows: intensive local computation is replaced by a generic COMPUTE routine (simple generic looping with the time estimated based on the original program). The input and output of data are eliminated (or modeled with some generic routines). The advantage of using the synthetic benchmark instead of the original benchmark is that it simplifies the evaluation process. In fact, in the process of designing and evaluating the algorithm for an application, the user can first construct the synthetic benchmark without actually specifying the details of the program, and the synthetic benchmark can be run on the machine to predict the performance of the algorithm. Poplawski further pointed out that this scheme can be used in an event-driven simulator where, executing the generic COMPUTE routine corresponds to advancing the clock based on the time COMPUTE spends. This speeds up the simulation considerably since the simulation of intensive computation is achieved by advancing the simulation clock, which is independent of the actual computation time.

The communication overhead, however, is not handled in Poplawski's framework. The message latency formula for the store-forward scheme under the contention-free assumption (refer to page 19) is used in [92]. One can not afford to do hop by hop detailed simulation for a large computation because of the long simulation time which depends on the message size. Using our formulae eliminates hop-by-hop simulation. This technique, combined with the generic COMPUTE, results in a scalable, efficient performance prediction scheme.

Conclusions

The contributions of this chapter are summarized as follows.

1. We study communication overhead issues related to the mapping problem. We directly measure the total completion time of two well-known benchmarks which are mapped with several mapping schemes with various interesting characteristics. Compared with previous work [23, 12, 1] which only measured network traffic performance, our results here offer direct insight into the impact of communication overhead, incurred by the topological mismatch, on actual performance.

2. To distinguish the effect of contention from that of the dilation, for each benchmark considered, we carefully choose two mappings so that the effects of these two factors can be clearly distinguished. Based on the simulation results, we are able to qualitatively characterize the effects on mapping performance. This extends Chittor's work [23] which did not clearly differentiate these two factors.

3. To better evaluate and characterize the performance of a mapping, we study message latency estimation using the message formulae developed in [76]. In this chapter, the message formulae are validated through simulation. Compared with message latency formulae developed in [28, 1], our formulae are sensitive to an application and are not based on statistical parameters for the network. The formulae have many applications which include analytical performance evaluation of a specific application, further analytical studies of communication overhead and mapping metrics, and simulation-based performance evaluation. In this chapter, we further develop a method to incorporate them into an event-driven multicomputer simulator.

4. We propose a general performance evaluation framework which integrates the techniques for message latency estimation in a simulator with the synthetic benchmark framework proposed by Poplawski [92]. The framework is scalable and practical.

We discuss future work in the following. For empirical studies of the mapping effect on actual benchmarks, more benchmarks are needed to better characterize a wide range of applications. For example, non-uniform message size distributions should be studied. Communication intensive and less intensive applications should be chosen for benchmarking.

Furthermore, other architectural models such as circuit-switching flow control schemes and virtual channel support should be developed. Important factors for application versus. architecture aspects such as communication/computation ratio should be also investigated. For communication overhead prediction, more benchmarking is needed to validate the message latency formulae. Further tradeoffs between accuracy and performance of a simulator should be studied.

The performance evaluation framework proposed offers a promising practical way for parallel program evaluation, which is much needed in the parallel computing community. Future work in this area involves the use of compile time data flow analysis to (semi)automatically generate the synthetic benchmark for a given program. Furthermore, we need to develop a comprehensive performance evaluation environment which can perform several levels of approximation based on compile time information as well as the user input. We believe such an environment will have important practical applications in parallel program performance evaluation.

CHAPTER IV

APPLICATION-SPECIFIC WORMHOLE ROUTINGS ON A MULTICOMPUTER NETWORK

Some multicomputers such as Meiko's transputers or Intel's iWarp machines allow the user to control the routing, in addition to the system supported default routing. Moreover, in many practical applications such as those in digital signal processing or in real-time environments, message passing structures can be known a priori. This provides an opportunity to optimize performance based on the knowledge of the applications. For message routing, instead of using a system-supported default routing scheme such as XY-routing on a mesh-connected multicomputer, we can design routing based on knowledge of the message passing requirements at compile time to reduce message traffic congestion. For example, consider a matrix transpose application where processor (i, j) needs to exchange data with processor (j, i). If we use XY-routing, it will cause unnecessary contention for some of the communication channels. On the other hand, a routing designed specifically for the messages can reduce the contention dramatically. Figure 19 shows the XY-routing of a 3×3 matrix transpose which has contention 2 on several channels. In contrast, we can route the message from $(1,3)$ to $(3,1)$ through $(1,3),(1,2),(2,2),(2,1),(3,1)$ and the message from $(3,1)$ to $(1,3)$ through $(3,1),(3,2),(2,2),(1,2),(1,3)$. This gives a contention-free routing.

As we pointed out in Chapter III, in a wormhole-routed multicomputer, mes-

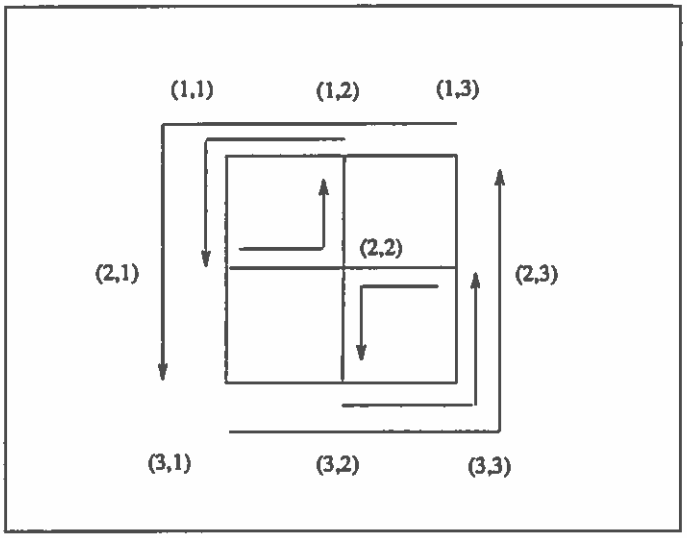


Figure 19: XY-Routing for 3×3 Matrix Transpose.

sage traffic congestion plays an important role in the performance of an application. Wormhole routing also introduces a new problem: deadlock can occur because blocked messages remain in the communication channels [30]. In this chapter, we develop a general framework to develop application-specific wormhole routings. Furthermore, based on the framework, we develop a method to generate message routes which have low message traffic congestion and are deadlock free.

Related Work

The problem of choosing paths for a given set of messages has been studied for various routing schemes. In [10], Bianchini and Shen propose a scheduling algorithm for message traffic in a multiprocessor network. The problem is formulated as a network flow problem based on the assumption that a message can be split into several small flows at a node. This assumption is not applicable to current tightly coupled, high-speed multicomputer networks since the overhead to manage message

splitting and combining may well exceed that of the whole message transmission through the network.

A scheduled routing framework is proposed by Shukla and Agrawal [111], for real-time periodical pipelining applications. In their scheme, the router of a processor sets up channel connections based on switch setting commands received from the application running in that processor. The switch setting commands are generated statically at compile time based on a priori knowledge of the task structure in a way that guarantees there will be an unobstructed path for each message during the whole message transmission period. Since messages are routed through the network without collision, the generated routing is always deadlock free. The scheme can only be used for an architecture which has specialized communication modules. Furthermore, it is not always possible to find paths such that messages do not collide. In this case, it is not clear how the deadlock is avoided.

The previous work which is most closely related to ours is the traffic routing scheme proposed by Kandlur and Shin [58]. They study the problem of choosing paths for an arbitrary set of messages to be routed in a network with virtual a cut-through routing capability. They show that the problem of choosing pairwise edge-disjoint paths for a given set of messages is NP-complete and an efficient heuristic algorithm is presented. The heuristic first randomly chooses a path for every message and then, tries to reroute one message at a time to decrease the *total cost*, which is defined as the summation (over all links) of the squares of the total message volume passing through each link. If the total cost can not be improved for all messages, the algorithm stops. It has been shown by simulation that the algorithm performs very well.

For a wormhole routed-network, however, Kandlur and Shin's algorithm may

generate a deadlocked routing. In Figure 20, the final routing of Kandlur and Shin's algorithm is optimal with respect to the cost function they define. However, the routing is deadlocked since in the example, the set of paths for m_1, m_2, m_3, m_4 create a channel dependency cycle $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4 \rightarrow C_1$. This is also true for M_1, M_2, M_3, M_4.

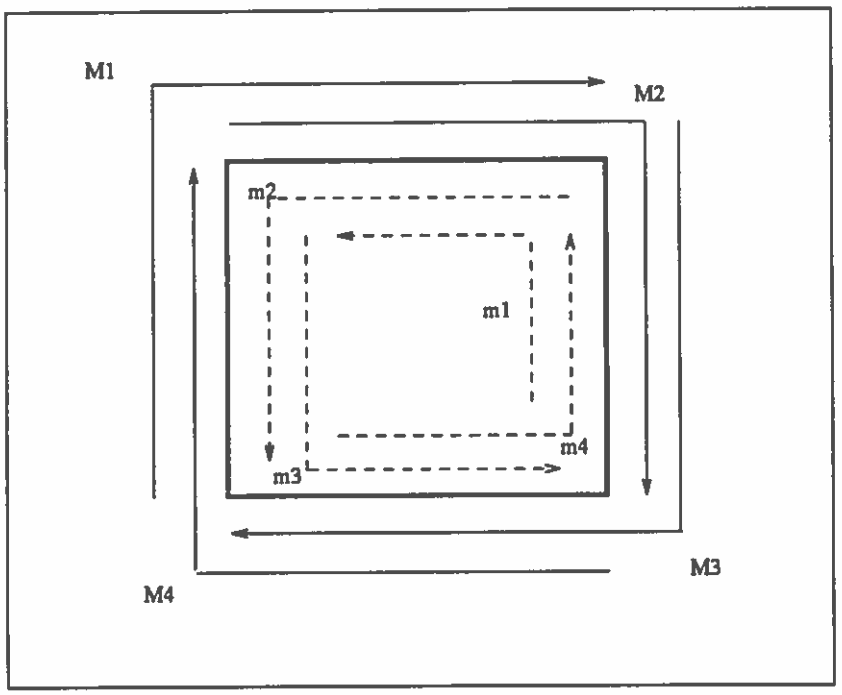


Figure 20: An Example Where Kandlur and Shin's Algorithm Generates a Deadlocked Wormhole Routing.

Problem Definition

By application-specific routing, we mean a routing designed specifically for the messages used in an application. This contrasts with the fixed schemes such as E-cube and XY routings which are independent of the messages to be routed. More precisely, we assume that processes of an application have been assigned to processors and we

also assume that message weights are known. Such weights can be the total volume of the messages sent between processors or the frequencies of message passing between two processors obtained by methods such as program profiling. In graph-theoretic terms, we formally define the problem as follows.

Definition 4.1

Let $A = (P, E)$ be a *directed* graph where nodes correspond to processors and edges to communication channels in the physical network. Let a set of messages be represented as $S = (M, W)$. M is a set of pairs of nodes in P where $(s, d) \in M$ represents the source processor and the destination processor for a single message, respectively; and W is the weight function which maps pairs in M to nonnegative integers representing message volume.

A *routing* is a function $\mathcal{R}$ which maps every pair $(s, d) \in M$ to a simple path $(p_0, p_1), (p_1, p_2), \ldots, (p_{k-1}, p_k)$ in A such that $p_0 = s, p_k = d$. □

As we point out in Chapter III, in wormhole routing, message traffic congestion is the predominant factor for performance. For the purpose of this chapter, we define contention metrics with respect to a specific routing.

Definition 4.2

For routing $\mathcal{R}$, the contention of a channel is defined as the number of the messages passing through it. The maximum contention of $\mathcal{R}$, $C(\mathcal{R})$, is the maximum contention over all channels. In particular, if the number of messages passing through a link h is $C(\mathcal{R})$, we call h a hot spot. If $C(\mathcal{R}) = 1$, we say $\mathcal{R}$ is contention-free. □

Intuitively, each channel whose contention equals $C(\mathcal{R})$ is one of the hottest traffic spots in the network. Hot spots are especially undesirable in a real-time application since the delay of messages may slow down the whole application and the deadlines may not be met. Our objective, in this chapter, is to find a routing which has minimal maximum contention.

Definition 4.3

The T-Cost of $\mathcal{R}$, $TC(\mathcal{R})$, is defined as

$$TC(\mathcal{R}) = \sum_{e \in E} (\sum_{m \in M, e \in \mathcal{R}(m)} W(m))^2$$

□

TC is the summation of the square of the *weighted channel contention* over all the channels. It reflects both the contention and dilation of the routing. The dilation effect is reflected since the longer the dilation of a message is, the more often its message size will appear in the summation. The contention effect is reflected by the square of the *weighted channel contention.* For example, suppose that there are only two messages m_1 and m_2 and $W(m_1) = W(m_2) = K$ where K is a constant. If the two messages do not have any overlapped path under routing $\mathcal{R}$, then $TC(\mathcal{R}) = K^2 \times (Dila(m_1)^2 + Dila(m_2)^2)$ where $Dila(m_1)$ and $Dila(m_2)$ are the *dilation* of m_1 and m_2 respectively (refer to Definition 2.1 on page 38). But if m_1 and m_2 have one overlapped channel in their routes, $TC(\mathcal{R}) = K^2 \times (Dila(m_1) + Dila(m_2)) + 2K^2$. Thus the contention effect is also taken into account.

TC was first defined and used as a cost metric in [58]. In the heuristic we propose here, we use this cost function as a secondary metric to control the algorithm

so that it does not detour messages too far to reduce the maximum contention.

In addition to the minimal maximum contention objective, the routing generated must be deadlock free in a wormhole routing. As we point out on page 26, deadlock can occur in a wormhole routing scheme when a circular channel waiting state is reached. A well-known condition for a deadlock-free wormhole routing is given by Dally [30] based on the concept of the channel dependency graph. The channel dependency graph $CDG_{\mathcal{R}}$ for a routing $\mathcal{R}$ is a directed graph whose nodes are channels used in the routing. If there is a path in $\mathcal{R}$ such that c_1, c_2 are two *successive* channels in this path, then there is an edge in $CDG_{\mathcal{R}}$ from node c_1 to c_2 (also called c_1 depends on c_2). It is clear that a channel dependency graph depends on a specific routing.

Property 1 *[30] Routing $\mathcal{R}$ is deadlock free for wormhole routing mechanism iff the channel dependency graph $CDG_{\mathcal{R}}$ is acyclic.*

Our optimization problem is to find a deadlock free routing which has the minimal maximum contention. Based on a result given in [58], which states that the decision problem of whether there exists a contention-free routing for an arbitrary set of messages in an arbitrary network is NP-complete and a contention-free routing is also deadlock free, we have

Theorem 4.1

The problem of deciding whether there exists a deadlock free contention-free routing for an arbitrary set of messages and an arbitrary network is NP-complete. □

Since the deadlock-free condition requires checking links in the original network A, it is useful to have a structure to represent link relationships explicitly. We define

a graph called the Generic Physical Channel Dependency Graph (GPCDG) for this purpose.

Definition 4.4

For a given network (directed graph) $A = (P, E)$, its GPCDG is a directed graph $D_A = (V, D)$ where its node set V is $E \cup P$ and its edge set D is defined as follows: for all $e_1 = (a, b), e_2 = (b, c) \in E$, we have an edge $(e_1, e_2) \in D$. Furthermore, for every edge $e = (p_1, p_2) \in E$, we also have an edge $(p_1, e), (e, p_2) \in D$ also. We call a node $l \in E$ a link node and a node $p \in P$ a processor node. □

Intuitively, a GPCDG D_A captures all possible dependencies among the physical channels of A and also retains the original network topology. A key difference between a channel dependency graph and a GPCDG is that the former corresponds to a specific routing and the latter is independent of any routing. Note the GPCDG is analogous to the *total graph* in graph theory [51] which is defined for an undirected graph. Figure 21 shows an example of GPCDG.

We now reformulate the optimization problem for a deadlock-free routing based on the GPCDG.

Problem Definition: Given a GPCDG D_A for a network $A = (P, E)$ and a set of messages $S = (M, W)$, a routing $\mathcal{R}$ is a function which maps each pair $(s, d) \in M$ to a simple path in D_A whose endpoints are s and d and whose interior points are link nodes in D_A. Furthermore, a routing $\mathcal{R}$ is deadlock-free iff the subgraph formed by message paths in GPCDG D_A is acyclic. The maximum contention becomes the maximum number of messages passing through a link node of the GPCDG. The objective of our optimization problem is to find a deadlock-free routing which minimizes

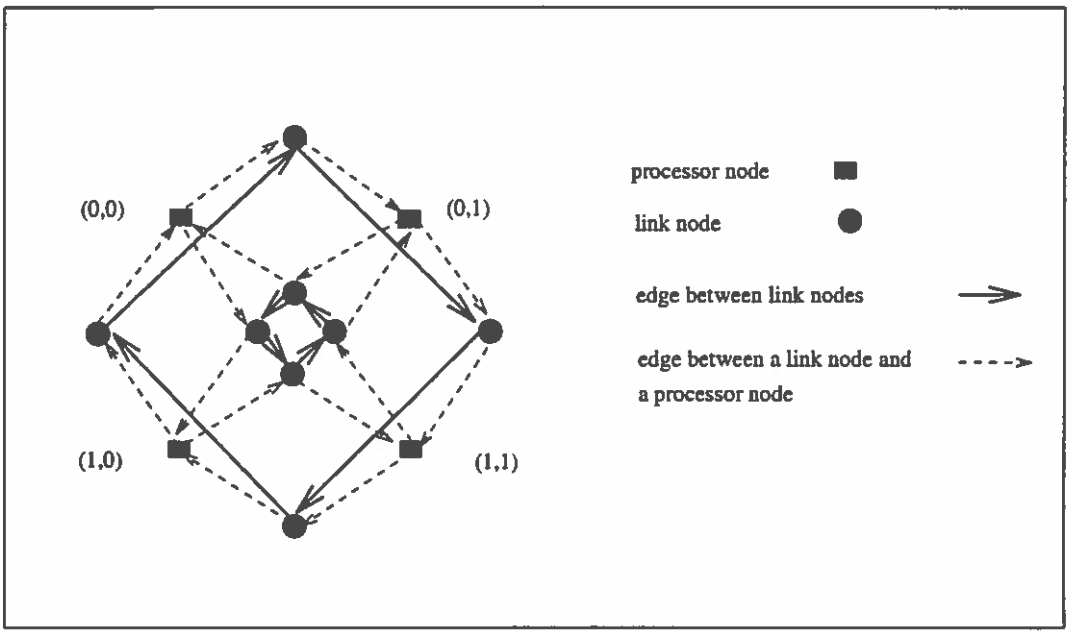


Figure 21: The GPCDG of a 2×2 Mesh.

the maximum contention over all link nodes in D_A.

Deadlock-Free Low-Maximum Contention Routing

Our deadlock-free heuristic (DFH) for the wormhole routing scheme is straightforward. It starts from an initially preselected *deadlock free* routing, iteratively tries to reduce the contention of hot spots (there might be several such spots) by rerouting messages through other channels with less contention such that the new routing is also *deadlock-free* and its T-Cost $TC(\mathcal{R})$ decreases. The algorithm stops if no further improvement can be achieved. Figure 22 shows the outline of DFH. DFH has two major procedures, *initialize* and *reroute*. Procedure *initialize* selects an initial deadlock-free routing. Procedure *reroute* tries to reroute a message through other channels other than the hot spot so that the T-Cost is decreased and no other hot spots through the rerouting are created. To efficiently find the hottest spots, a prior-

```
/*DFH */

R = initialize(D_A);
repeat {
        TC_old = TC(R);
        for (every hottest spot h in R){
            if (there exists a message m passing through h which
              can be rerouted through channels other than h and
              the T-Cost decreases) {
              R=reroute(m); }}}
until {TC_old == TC(R)};
```

Figure 22: Outline of the Deadlock-Free Heuristic DFH.

a priority queue is used to manage link nodes with priorities being their contentions. The following two subsections describe the two routines in more detail.

Initialization

Finding a deadlock-free routing as the initial routing for DFH is straightforward for many regular networks such as hypercubes, meshes and tori. For a hypercube, we can use E-cube routing which always routes a message in the order of decreasing dimensions. This routing has been proved to be deadlock-free for wormhole routing [30]. For a 2D mesh or torus, we can use XY routing which always routes a message first along the X-direction and then along the Y-direction. This is also known to be deadlock-free. Some applications, however, may use irregular interconnections and there may be no known deadlock-free fixed routing for such networks. For example, the iWarp system allows the user to configure an irregular network by setting up so-called pathways [14]. We propose a simple method to find an initial deadlock-free routing for an irregular network topology. The method uses a spanning tree of the network to find such a routing.

Property 2 *Let $A = (P, E)$ be a bidirectional network, which is represented as an undirected connected graph and $S = (M, W)$ be a set of messages to be routed, it is always possible to find a deadlock-free wormhole routing $\mathcal{R}$ for S in A.*

Proof

Let ST be a spanning tree of A. ST can be found efficiently (for example, with using a depth-first search). Let DST be the directed graph generated from ST by replacing a single undirected edge (p_1, p_2) in ST with two directed edges (p_1, p_2) and (p_2, p_1). We define a routing $\mathcal{R}$ as follows. Since for any two nodes $(s, d) \in M$, there exists a unique simple path $P = (p_1, p_2, \ldots, p_n)$ from s to d $(s = p_1, d = p_n)$ in DST, we designate the directed path of P (which exists in A since A is bidirectional) as the routing path for (s, d).

To prove $\mathcal{R}$ is deadlock-free, assume the opposite. Then by Property 1, the channel dependency graph $CDG_{\mathcal{R}}$ has a simple cycle $c_1, c_2, \ldots, c_n$ where c_i corresponds to edge (s_i, d_i) in DST for $i = 1, \ldots, n$. Thus $d_i = s_{i+1}$ for $i = 1, \ldots, n-1$ and $d_n = s_1$ and $s_1, \ldots, s_n$ form a cycle C in DST. But the only cycles in ST are those which are formed by cycles (p_1, p_2) and (p_2, p_1). Therefore, there exists an $1 \leq i < n$ such that $e = (s_i, s_{i+1})$ and $e' = (s_{i+1}, s_i)$ are two successive edges in C. But this implies that e depends on e' and e' depends on e, which is not correct since $\mathcal{R}$ always routes messages through a simple path in ST. □

The importance of the above property is that it indicates there is *always* a deadlock-free routing for any bidirectional network. Furthermore, the spanning tree approach efficiently finds an initial deadlock-free routing (with linear time complexity in the

number of edges of A). The drawback of the algorithm is that the message traffic contention of the resultant routing is high because of the tree structure. Thus, when a network is regular, we use the fixed routing as its initial routing.

Reroute a Message away from a Hot Spot

Procedure *reroute* takes a routing $\mathcal{R}$, a hot spot h in GPCDG D_A, and a message m passing through link h as its inputs and returns a new routing. Assume the nodes immediately before h and immediately after h in the path for message m are a and b respectively (note: a, b may be processor nodes). *reroute* tries to reroute m from a to b by trying to avoid passing through h. It should be noted that we can not simply reroute m through any path from a to b even though such rerouting can decrease maximum contention and the T-Cost. This is because such a path may create a deadlock. To simplify the discussion, we need the following definitions.

Definition 4.5

> For a routing $\mathcal{R}$, $\mathcal{R}$ is also used to denote the channel dependency relation induced by $\mathcal{R}$ on link nodes in D_A. A path is said to be *deadlock-free with respect to a routing* $\mathcal{R}$ if adding the dependencies in the path to $\mathcal{R}$ still preserves the acyclic condition in D_A. □

Let $\mathcal{R}_1$ correspond to the channel dependency relation after the removal of message m from path a, h, b. Note that $\mathcal{R}_1$ may still retain dependency $a \rightarrow h$ or $h \rightarrow b$ since there might be other messages routed through these links. Also note if $\mathcal{R}$ is deadlock-free, $\mathcal{R}_1$ is deadlock-free too, since removal of dependencies can not introduce deadlock. The goal of *reroute* is to find a deadlock-free path from a to b to replace a, h, b such that the T-Cost is decreased.

The algorithm uses depth-first search to find such a path. It starts with an initial partial path containing only a and examines the unsearched link nodes adjacent to the most recently added node of P. If adding an unsearched node does not cause deadlock and the new partial T-Cost does not exceed the old T-Cost, then the node is marked as searched and is added to P. The procedure continues until it either reaches b or no new node can be found which satisfies the deadlock-free condition and decreases T-Cost. In the former case, a new route from a to b has been found. In the latter case, the algorithm backtracks to the previous node in P. If no new deadlock free path is found (i.e., if a is examined again), then m is still routed through a, h, b.

Figure 23 shows the algorithm. *Ocost* is the T-Cost of the initial routing $\mathcal{R}$ and *Pcost* is the T-Cost of the current partial routing.

```
/* reroute m from h. */
/* Assume m passes link a, h, b successively.*/
P = {a}; mark(a);
Ocost =cost(R);
Pcost=cost(remove-m-from-h(R, m, h));
while (P not empty and last(P) ≠ b) {
      if (there is a deadlock-free unmarked node u adjacent to last(P)
        and update(Pcost, u) < Ocost){
        P = P⋃{u};
        mark(u);
        Pcost=update(Pcost, u) }
        else
              {P = P−{last(P)};
              recover(Pcost);}}
```

Figure 23: *reroute* Function.

Procedure *reroute* requires testing whether adding a node causes deadlock in $P \bigcup \mathcal{R}$. To efficiently accomplish this, we maintain the *transitive closure* of the chan-

nel dependency relation induced by $\mathcal{R}$. This is because to test whether there is a dependency from node n_1 to node n_2, one would have to test whether there is a path from n_1 to n_2 in the graph induced by $\mathcal{R}$. The transitive closure of $\mathcal{R}$ reduces the complexity of this testing to be a constant.

Let $\mathcal{R}^*$ be the transitive closure of $\mathcal{R}$ and $\mathcal{R}_1^*$ be the corresponding transitive closure of $\mathcal{R}_1$. Since computing the transitive closure [2] is relatively expensive ($O(N^3)$ where N is the number of link nodes), we want to avoid recomputing the entire new transitive closure every time we find a new path to replace path a, h, b. This can be achieved by maintaining the transitive closure as a matrix whose ij entry represents the number of directed paths from a link node i to another link node j instead of simply a binary entry to indicate whether there is a path from i to j. Each time a new path is found to replace a, h, b, only some entries of the transitive closure are changed. The time it takes to compute the number of paths for all pairs of link nodes after the initial routing has been chosen is the same as to compute the standard binary transitive closure. Figure 24 shows the algorithm to update the transitive closure once a new path has been found. The time complexity to update the transitive closure is reduced to $N^2|P|$, where $|P|$ is the length of P.

The algorithm to test whether a node is deadlock-free or not when P is added to $\mathcal{R}_1$ is shown in Figure 24. Since it is possible that no new path other than a, h, b can be found, we would like to retain $\mathcal{R}$ and not update it until a new path is found. The algorithm in Figure 24 uses $\mathcal{R}$ instead of $\mathcal{R}_1$ to accomplish the deadlock testing. This is justified in Property 3.

Property 3 *Path P, as chosen by algorithm* reroute, *is deadlock-free with respect to $\mathcal{R}_1$.*

```
/* update-transitive-closure(R*, P, a, h, b)
returns an update R1* by replacing a, h, b with P in R* */
R1* = R*;
/* removal of a → h (if any) */
if {removing m from path a, h, b causes a ↛ h}
  for {all l such that l → a or l = a }
      for {all k such that k → a or k = h }
          R1*(l, k) = R1*(l, k) − 1;
/* similar code for the removal of h → b and adding P, omitted*/


%deadlock-free-test(u, P, R*, b)

if    {there is no p ∈ P such that uR*p and not(bR*u) }
      return true;
else  return false
```

Figure 24: Updating Functions for a Transitive Closure and *deadlock-free-test*.

Proof

We first prove that P is deadlock-free with respect to $\mathcal{R}$. Notice P defines a total order $<$ (i.e., the successive order of edges in P). If adding P to $\mathcal{R}$ causes a simple cycle C, C must contain some subpaths in P. Figure 25 shows the structure of C where P_i $(i = 1, \ldots, m)$ are subpaths of P in C. Otherwise, C is a cycle in $\mathcal{R}$, which is a contradiction. Let h_i and t_i be head and tail of P_i respectively.

If there exists an i such that $t_1 < h_2, t_2 < h_3, \ldots, t_{i-1} < h_i$ and $t_i \not< h_{i+1}$, we have $h_{i+1} < t_i$. From the structure of C, we notice that there exists a path L in $\mathcal{R}$ (refer to Figure 25) from t_i to h_{i+1}. But this is a contradiction since the algorithm *deadlock-free-test* guarantees that there is no such a path in $\mathcal{R}$.

If there is no such i, then $h_1 < t_1 < t_m$ and there exists a path in $\mathcal{R}$ from t_m to h_1. Based on the similar argument, this is impossible.

Since $\mathcal{R}_1$ is contained in $\mathcal{R}$. P must be deadlock free with respect to $\mathcal{R}_1$. □

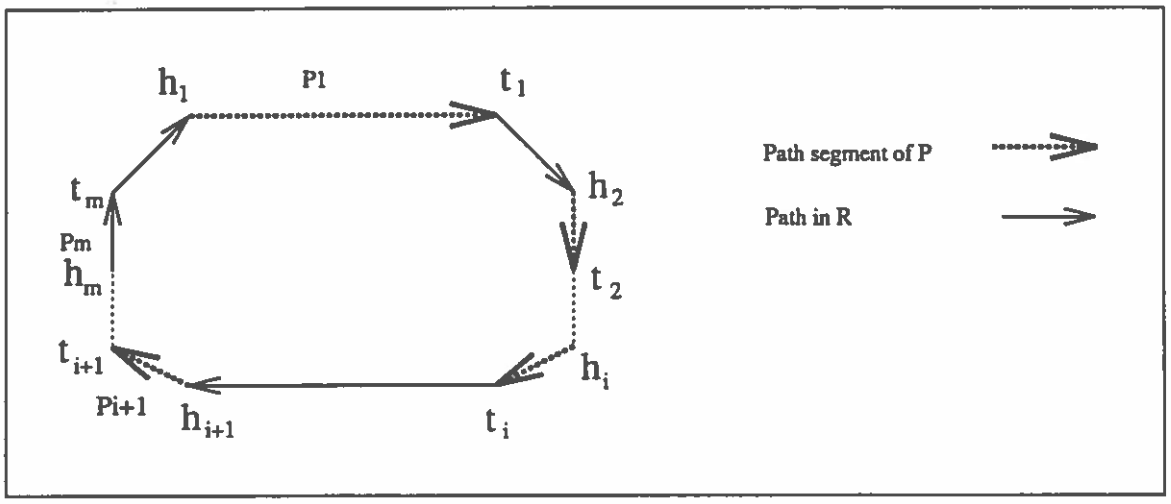


Figure 25: Illustration of the Cycle C for the Deadlock-Free Testing.

Since *deadlock-free-test* is based on $\mathcal{R}$ rather than $\mathcal{R}_1$, a natural question is whether it eliminates some paths which are deadlock-free with respect to $\mathcal{R}_1$ but not with respect to $\mathcal{R}$ (it is possible since $\mathcal{R}$ contains $\mathcal{R}_1$). The following property answers this question.

Property 4 *A deadlock-free path $P = a_0, \ldots, a_k$ where $a = a_0$ with respect to $\mathcal{R}_1$ is also deadlock-free with respect to $\mathcal{R}$.*

Proof

If adding P to $\mathcal{R}$ causes a simple cycle C, then $\mathcal{R}$ must have more dependencies than $\mathcal{R}_1$. It is safe to assume that $\mathcal{R}$ has both $a \rightarrow h, h \rightarrow b$.

Based on the similar argument in the proof of Property 3, there exists a situation where there are $a_1, a_2 \in P$ such that $a_1 <_P a_2$ and there is a path L in $\mathcal{R}$ from a_2 to a_1.

If there is no $a \rightarrow h$ and $h \rightarrow b$ in L, then, C is a cycle in the channel dependency graph $\mathcal{R}_1 \bigcup P$, which is a contradiction.

Therefore, there exists $a \rightarrow h$ or $h \rightarrow b$ in L. If there exists $a \rightarrow h$, then the path segment from a_2 to a does not have any $a \rightarrow h$ and $h \rightarrow b$ since C is simple. Based on the previous argument, we know this is impossible.

The only remaining case is when there is only an $h \rightarrow b$ in L, in addition to other dependencies in $\mathcal{R}_1$. But since for a_1 to be in P, *deadlock-free-test* must have guaranteed that there is no path from b to a_1 in $\mathcal{R}$, which includes $h \rightarrow b$. This is clearly a contradiction. Hence, we prove the property. □

An upper bound of the time complexity of DFH can be estimated as follows. Procedure *initialize* takes time $O(|E|)$ if the spanning tree approach is adopted and takes $O(1)$ if a default routing is used. Calculating the initial transitive closure takes $O(|E|^3)$. Since there are at most $|M|$ number of messages, there are at most $|M|$ different maximum contention. Also, there are at most $|E|$ link nodes whose contention is the same. Hence, the total time is at most $O(|E| + |E|^3 + |M||E|(T(\textit{reroute})))$, where $T(\textit{reroute})$ is the time complexity for *reroute*. Since at most $|E|$ edges are examined in *reroute* and *deadlock-free-test* takes at most $|P|$ (P is the path considered) steps, we have $T(\textit{reroute}) = O(|E||P| + |E|^2|P|)$. Since we use T-Cost to control the path length, we can easily derive that $|P| = O(|M|W)$ where W is the maximum weight of messages in M. Thus, the total complexity of DFH is $O(|M|^2|E|^3W)$.

Performance

Two test suites are used to evaluate DFH. Each is evaluated on the torus and hypercube topologies. In both tests, performance is evaluated using both maximum contention and T-Cost. The performance of DFH is compared with those of the E-cube and XY fixed-routing schemes respectively. These fixed-routing schemes are also used as the initial deadlock-free routings.

The first test suite includes two types of randomly generated message distributions. In the first case, messages are uniformly distributed. Three independent uniform random number generators are used to generate source nodes, destination nodes and message weights respectively. Message weights range from 1 to 50. The number of messages ranges from 10 to 200.

In the second case, messages are nonuniformly distributed on the network in a way to capture communication locality which is usually exhibited after the assignment of processes to processors for an application. This is modeled by dividing the network into four equal parts and messages are distributed uniformly in two quadrants and no messages exist in the other two quadrants. The topologies are a 6×6 torus and a 5-dimensional hypercube. A 5-dimensional cube is divided into four subcubes labeled as $00xxx, 01xxx, 10xxx, 11xxx$ where x represents a don't care bit and xxx represents a subcube of dimension 3. Messages are uniformly distributed in subcubes $00xxx$ and $11xxx$. No messages exist in subcubes $01xxx$ and $10xxx$. A 6×6 torus is divided into four 3×3 meshes with additional wrap around links. In the upper left and the lower right quadrants, messages are uniformly distributed and no messages exist in the upper right and lower left quadrants. The same message weight ranges and the number of messages, as in the first case, are used.

For both cases, performance metrics are averaged over 25 runs of DFH for a given number of messages. It is observed that the standard deviation of the mean maximum contention for all data is less than 6%.

Figure 26 compares the maximum contention of the routing generated by DFH with that of E-cube routing. Figure 27 shows the percentage improvement of DFH over E-cube for both maximum contention and the T-Cost. For nonuniform case, DFH consistently reduces maximum contention by 30% to 40% and for the uniform case, by 15% to 25% compared with E-cube routing. The T-Cost has much less percentage improvement than maximum contention. This is because DFH takes the maximum contention as its primary optimization goal.

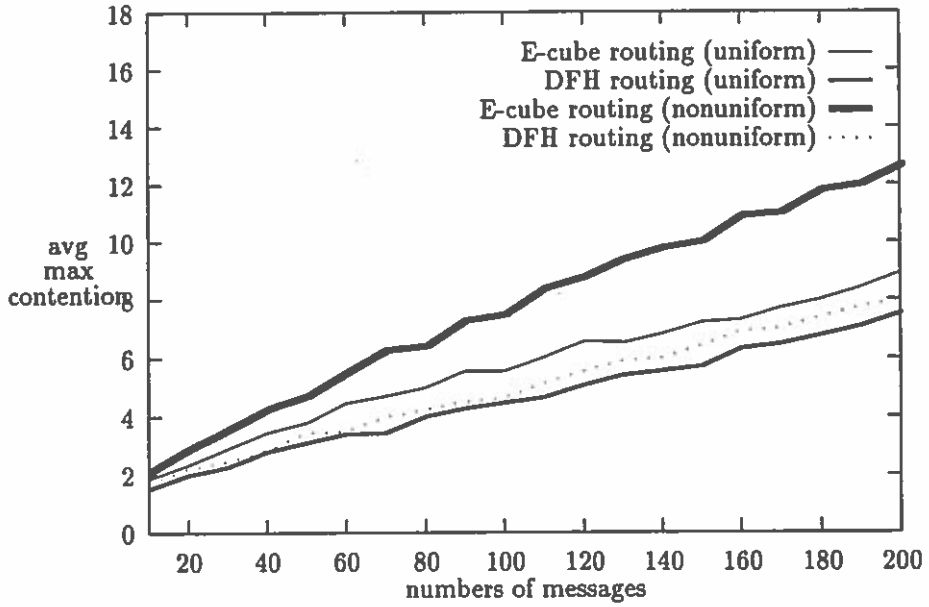


Figure 26: Average Maximum Contention for a 5-Dimensional Cube for Both Uniform and Nonuniform Message Distribution.

Figure 28 and Figure 29 show the maximum contentions for DFH routing and XY routing and the percentage improvement for both maximum contention and T-

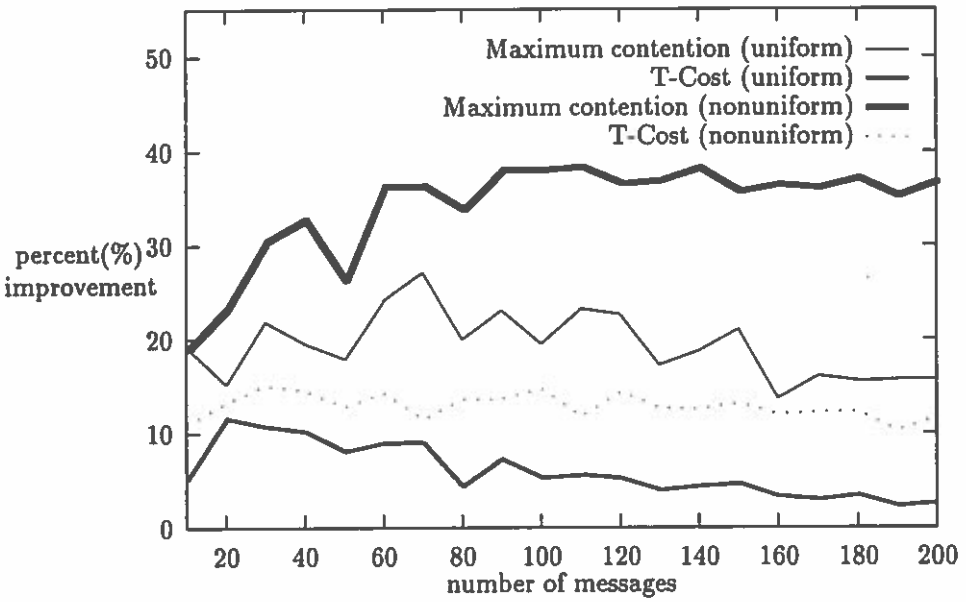


Figure 27: Percentage Improvement of Maximum Contention and T-Cost for a 5-Dimensional Cube for Both Uniform and Nonuniform Message Distribution.

Cost on a 6×6 torus. Again, for the percentage improvement, we see a fairly good gain for maximum contention and a moderate gain for T-cost. It can be seen also that the improvement is less than that in the binary cube case for the same number of messages. This is because the binary cube has more connectivity than the torus.

The second test suite includes two specific applications. For both applications, a simple greedy embedding algorithm is used to assign processes to target topologies. The first application is the n-body algorithm which is designed for the Caltech Cosmic Cube [5]. Figure 30 gives a description of the problem. In the simulation for the 15-body problem both 5×3 torus and 4-dimensional cube topologies are used.

The second application is a program called AVHTST which is used to determine cloud properties from satellite imagery data [104, 57]. Figure 31 gives a description of the problem and its task structure. In the simulation, an 18 nodes AVHTST program is routed on a 4×4 torus and a 4-dimensional cube respectively.

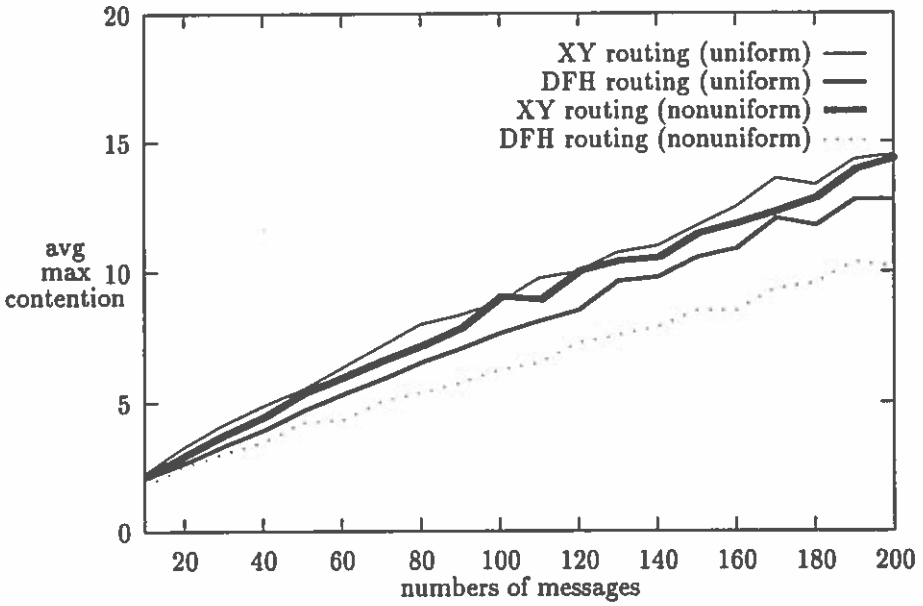


Figure 28: Average Maximum Contention for a 6×6 Torus for Both Uniform and Nonuniform Message Distribution.

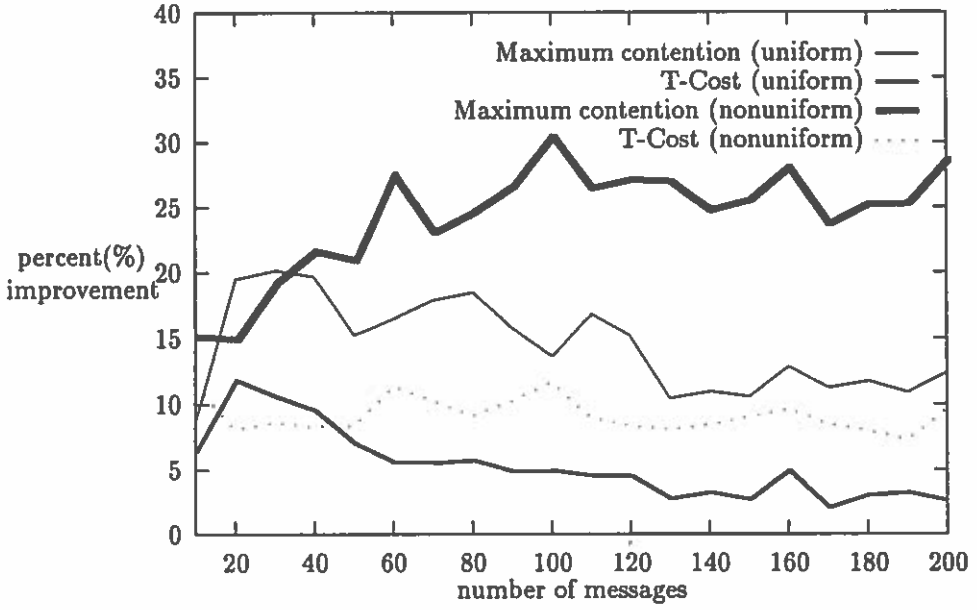
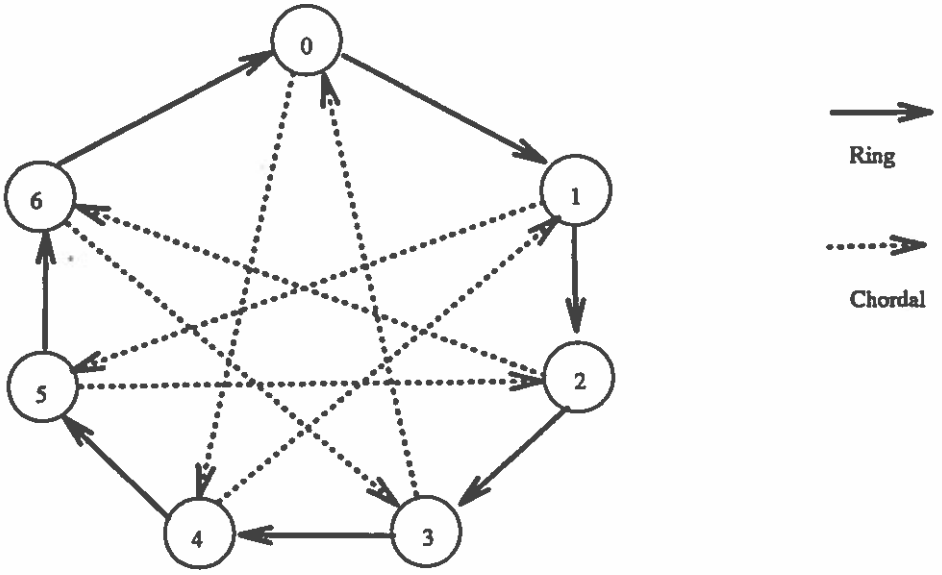
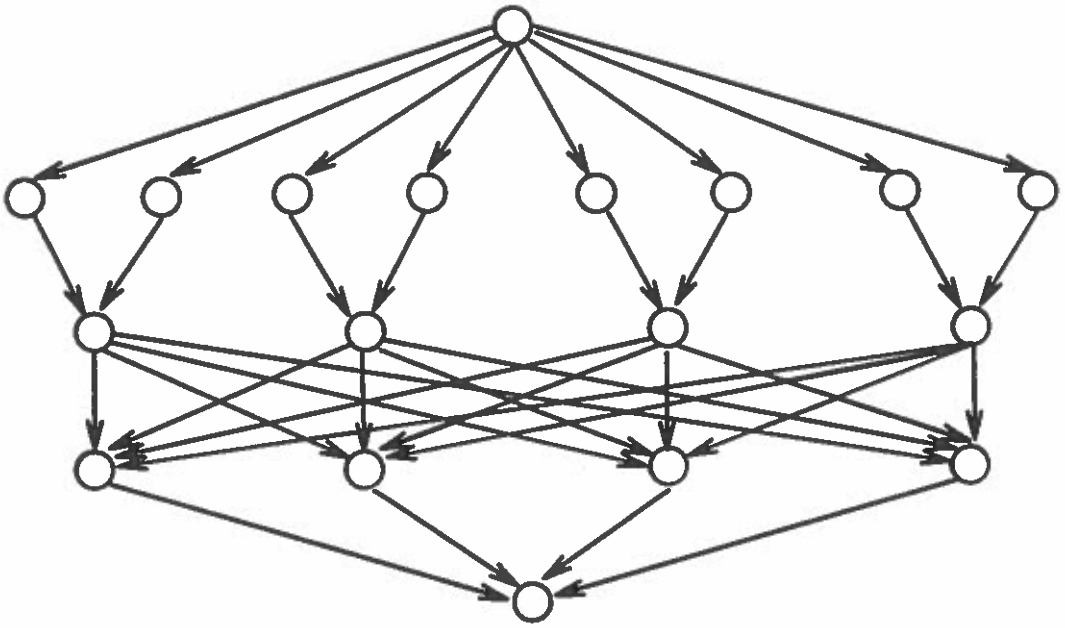


Figure 29: Percentage Improvement of Maximum Contention and T-Cost for a 6×6 Torus for Both Uniform and Nonuniform Message Distribution.



The *n-body problem* requires determining the equilibrium of n bodies in space (where n is odd) under the action of a (gravitational, electrostatic, etc.) field. This is done iteratively by computing the net force exerted on each body by the others (given their "current" position), updating its location based on this force, and repeating this until the forces are as close to zero as desired. The parallel algorithm presented by Seitz [5] uses Newton's third law of motion to avoid duplication of effort in the force computation. It consists of n identical processes, each one responsible for one body. The processes are arranged in a ring and pass information about their accumulated forces to its neighbor around the ring. After $(n-1)/2$ steps, each task will have received information from half of its predecessors around the ring. Each task then acquires information about the remaining bodies by receiving a message from its chordal neighbor halfway around the ring. This is repeated to the desired degree of accuracy. In the above is the task graph of 7-body problem.

Figure 30: The Description of n-Body Problem and Its Task Graph.



AVHTST is part of a program for the study of cloud properties based on satellite imagery data. It is responsible for data analysis and reduction. Input to *AVHTST* consists of c channels of data, where each channel is represented by a pixel array corresponding to a scene of data at a distinct wavelength. Furthermore, each array is divided into $f \times f$ frames. Each frame is first processed independently and then the results for the corresponding frames in all c channels are combined. Finally, the combined results for $f \times f$ physical frames are broadcast to all other frames for an adjustment of local results. In the last step, the results are aggregated. Above is the task graph for *AVHTST* with 2 channels of data each with 2 by 2 frames per scene. Nodes are labeled with a triple where the first component represents the channel number and the remaining components are the frame coordinates. All edges have unit weight.

Figure 31: The Description of AVHTST Benchmark and Its Task Graph.

Table 11 shows the simulation data for the two applications. In all cases, there is significant improvement with respect to maximum contention. In addition, all percentage improvements for T-Cost are above 10%.

Table 11: Performance of DFH for the Applications

Applications	Max. Contention			T-Cost		
	Fixed	*DFH*	Percentage	Fixed	*DFH*	Percentage
15-body on 5×3 torus	5	3	40.0%	27800	20800	25.2%
15-body on 4-dim cube	3	2	33.3%	23400	19000	18.8%
AVHTST on 4×4 torus	7	5	28.6%	410	367	10.5%
AVHTST on 4-dim cube	5	3	40.0%	198	166	16.2%

We observe that in the simulation, the number of hot spots is usually much greater than one and as the number of contending messages decreases, the number of links which have that number of contending messages increases. This explains the fact that when the maximum contention is small, it is extremely hard to further reduce it.

The performance results conform with our intuition. When messages are uniformly distributed on the network, commonly-used fixed-routings may evenly load network channels. As a result, it is difficult for DFH to improve the T-Cost. For the cases where messages are nonuniform distributed or for specific applications, since message distribution often exhibits spatial locality, the fixed-routing schemes are more likely to yield higher contention. Thus, DFH can have more improvement for both maximum contention and T-Cost.

Conclusions

The deadlock avoidance problem, which arises from wormhole routing, poses some challenging problems in the design of application specific routings. More sophisticated techniques need to be developed. Since channel dependencies cause deadlock, we formulate the problem of finding a low maximum contention deadlock-free routing as a graph theoretic problem on the Generic Physical Channel Dependency Graph of the original network. A simple heuristic is proposed. To avoid expensive updating of the channel dependencies, a modified transitive closure algorithm is used. The performance of the heuristic is studied. The algorithm converges very fast in all the testing sets. The heuristic has significant improvement for nonuniform message distribution as well as for two specific applications and has moderate improvement for uniformly distributed messages over well known fixed routing schemes.

The application-specific wormhole routing generated by our heuristic can be used in advanced multicomputers. An example is Intel's iWarp multiprocessor system [14]. In an iWarp system, the physical interconnection network is a torus. However, the interconnection can be logically reconfigured by setting up *pathways*. A default routing called *street sign routing*, which is essentially the XY routing scheme, is supported by the system. However, applications can change routes by generating necessary routing information in the header of a message.

Future work includes developing more efficient heuristics for application-specific deadlock-free routing. We should further study how to fully utilize the regularity of the GPCDG for a regular network such as a mesh or a hypercube to develop more efficient heuristics. Furthermore, as is pointed out in Chapter III, we should study the problem of how to reduce message traffic by reducing path level contention instead of

maximum traffic contention. We should also study how to take into account temporal information in an application when developing a routing algorithm. For the TCG model, one simple way to extend the work in this chapter is to apply the application-specific routing phase by phase, i.e., we can generate routings for each phase instead of for the whole static task graph.

CHAPTER V

AN EFFICIENT HEURISTIC FOR APPLICATION-SPECIFIC ROUTINGS ON A MESH CONNECTED MULTICOMPUTER

In Chapter IV, a method is proposed to generate deadlock free low traffic contention routings for applications. The method does not assume that there is virtual channel support in the router. In this section, we study the problem of generating an application-specific routing for a mesh connected multicomputer with virtual channel capability.

Given a mesh with virtual network support, we study the problem of finding a minimum maximum contention routing for a given application whose message passing structure is known a priori. The routing on the virtual networks is called minimal routing since all feasible paths are of shortest Mahattan distance.

Related Work

Besides the work mentioned on page 80, there has been extensive study on off-line permutation routing algorithms (refer to page 24) on mesh-connected multicomputers. The pioneer sorting algorithm on mesh was proposed by Thompson and H. T. Kung [114] and an optimal routing algorithm was proposed by Nassimi and Sahni [86]. Recently, permutation routing algorithms with constant queue size on a mesh have also been proposed by Leighton et. al. [72], Krizanc et. al. [65], and Rajasekaran et al. [94]. All of these work, however, assume that a routing is a

permutation.

Minimal adaptive routers have been also proposed by Konstantinidou [62] and Gravano et al. [47]. The algorithms, however, are designed specifically for a distributed, on-line router.

Application-Specific Routing on a Partitioned Mesh

For the purpose of this chapter, we consider our target architecture to be an n dimensional mesh which is partitioned into virtual networks as described in Chapter II. Formally, we define a routing on a *partitioned* mesh as follows.

Definition 5.1

Let $A = (P, E)$ be a directed graph representing an n dimensional mesh where nodes correspond to processors and edges to communication channels in the physical network. Let A be partitioned into 2^{n-1} virtual networks as discussed on page 26 . Let a set of messages be represented as M which is a *multiset* of pairs of nodes in P, where $(s, d) \in M$ represents the source processor and the destination processor, respectively, for a single message . A *routing* on A is a function $\mathcal{R}$ which maps every pair $(s, d) \in M$ ($s = (s_1, \ldots, s_n)$ and $d = (d_1, \ldots, d_n)$) to a simple path $P = (p_0, p_1, p_2, \ldots, p_k)$ in A such that $p_0 = s, p_k = d$ and the length of P is $k = \sum_{i=1}^{n}(|s_i - d_i|)$. Such a routing is called a *minimal* routing in [15] and is abbreviated as *m-routing.* □

It is possible that M is not a permutation. Furthermore, two messages with the same source and same destination addresses may be present in M (i.e., M is a multiset).

For a minimal routing, message collision is the predominant factor for performance. This is because all messages are routed through some *shortest* path and the effect of distance for the routing performance is minimized.

As in Chapter IV, our objective, in this chapter, is to find a routing which has minimal maximum contention which is defined in Definition 4.2 in Chapter IV.

Definition 5.2

Optimization Problem: Given an n-dimensional mesh A, of size $m_1 \times m_2 \times \ldots \times m_n$, and a set of messages M to be routed, find a m-routing $\mathcal{R}$ such that $C(\mathcal{R})$ is minimal.

The Contention-Free Decision Problem (CFD): Is there an m-routing $\mathcal{R}$ such that $C(\mathcal{R}) = 1$ for a given set of messages M. □

We show, in the following, that the optimization problem is NP-hard for an n-dimensional mesh when $n > 2$. In particular, we show that the decision problem can be polynomially reduced to the 3-Sat problem [41]. The proof is based on the method used in [58] where the problem of finding a contention-free routing (not necessarily minimal) on an *arbitrary* network (graph) is proved to be NP-complete.

Theorem 5.1

The contention-free decision (*CFD*) problem is NP-complete for $n > 2$.

□

Proof

We show that the problem is NP-complete for three-dimensional meshes. The conclusion for higher dimension can be deduced easily.

The problem is clearly in NP since given a guess of a possible routing, we can easily verify whether the routing is contention-free or not in polynomial time. We show that 3-Sat(isfiability) problem [41] is polynomially reducible to *CFD*.

Let Θ be a proposition which is a conjunction of k disjunctive clauses with a total of m variables $x_1, \ldots, x_m$ in a 3-Sat problem. We construct a routing problem instance R_P on a three dimensional mesh A such that R_P has a contention-free m-routing iff Θ is satisfiable.

We construct, in A, a subgraph A_1 which is crucial for the construction of R_P.

Figure 32 shows the construction of A_1. We denote a node in A by its coordinate (x, y, z) where the first node is $(1, 1, 1)$. Corresponding to the m variables, we construct m rectangles in plane $z = k+1$. Each rectangle represents the occurrences of one variable in Θ. The size of the i-th rectangles is $2L_i \times 2$ where L_i is the number of occurrences of both literal x_i and $\bar{x}_i$. Two corner nodes in the i-th rectangle are denoted as C_1^i, C_2^i respectively where, in the rectangle, C_1^i, C_2^i are the closest and farthest points to origin $(1, 1, 1)$ respectively. We call nodes with smaller y coordinates lower trail nodes and nodes with larger y coordinate upper trail nodes in a rectangle. We further use U_j^i to denote the node whose x-coordinate is the j-th smallest among all possible x coordinates in the upper trail of the i-th rectangle. In the same way, we label the j-th lower node as V_j^i. Notice that $C_1^i = V_1^i, C_2^i = U_2^{2*L_i}$. We intend to relate the jth-occurrence of x_i in Θ to edge $(U^i_{2*(j-1)+1}, U^i_{2*(j-1)+2})$ or edge $(V^i_{2*(j-1)+1}, V^i_{2*(j-1)+2})$,

depending on whether x_i or $\bar{x}_i$ occurs. These rectangles are connected by a horizontal edge which connects C_2^i and C_1^{i+1}. Furthermore, we designate two special nodes S_F and D_F which will be used as the source and destination nodes for a message. We connect S_F to C_1^1 and, C_2^m to node D_F horizontally. Figure 32 shows the coordinates of all of nodes we have defined in A_1.

For the i-th clause ($i = 1, \ldots, k$), we designate two nodes S_i, D_i which are intended to be source and destination nodes for a message. S_i is placed in plane $z = i$ which is under plane $z = k + 1$. D_i is placed in plane $z = k + i + 1$, which is over plane $z = k + 1$. Their coordinates are shown in Figure 32. Furthermore, we connect S_i and D_i by paths which pass through rectangles as follows. If the j-th occurrence of literal x_p or $\bar{x}_p$ is in the i-th clause C, then we construct a path from S_i to D_i by joining the follow path segments: S_i to $I_p = (2 + \sum_{u=1}^{i-1} L_u + 2 * (j-1), 1, i)$ in plane $y = 1$, notice that the x-coordinate of I_p is the same as the x-coordinate of $U^i_{2*(j-1)+1}$ or $V^i_{2*(j-1)+1}$; I_p to $J_p = I_p + (0, \delta + p, 0)$ where $\delta = 1$ if this occurrence is x_p otherwise $\delta = 2$; J_p to $K_p = J_p + (0, 0, k - i + 1)$; K_p to $K'_p = K_p + (1, 0, 0)$; Here, K_p and K'_p are $U^i_{2*(j-1)+1}, U^i_{2*(j-1)+2}$ or $V^i_{2*(j-1)+1}, V^i_{2*(j-1)+2}$, depending on δ. K'_p to $Q_p = K'_p + (0, 0, i)$; Q_p to $R_p = Q_p + (0, m + 3 - \delta - p, 0)$; R_p to D_i. Here, δ is chosen such that, when the occurrence is x_p, the path passes through an edge in the lower trail of the rectangle and when the occurrence is $\bar{x}_p$, the path passes through an edge in the upper trail.

It can be verified that paths from S_i to D_i are disjoint from paths from S_j to D_j in A_1 for any $i \neq j$.

The largest coordinates in A_1 are from $D_k = (2 + \sum_{i=1}^{m} L_i, m+3, 2k+1)$. We choose the size of A as $(2 + \sum_{i=1}^{m} L_i) \times (m+3) \times (2k+1)$.

The routing problem R_P is constructed as follows: the message set M consists of a message from S_F to D_F, a message from S_i to D_i for $i = 1, \ldots, k$ and messages from a to b for any edge (a, b) in A but *not* in A_1 (we consider A to be a directed graph and every edge in a mesh corresponds to two directed edges in A).

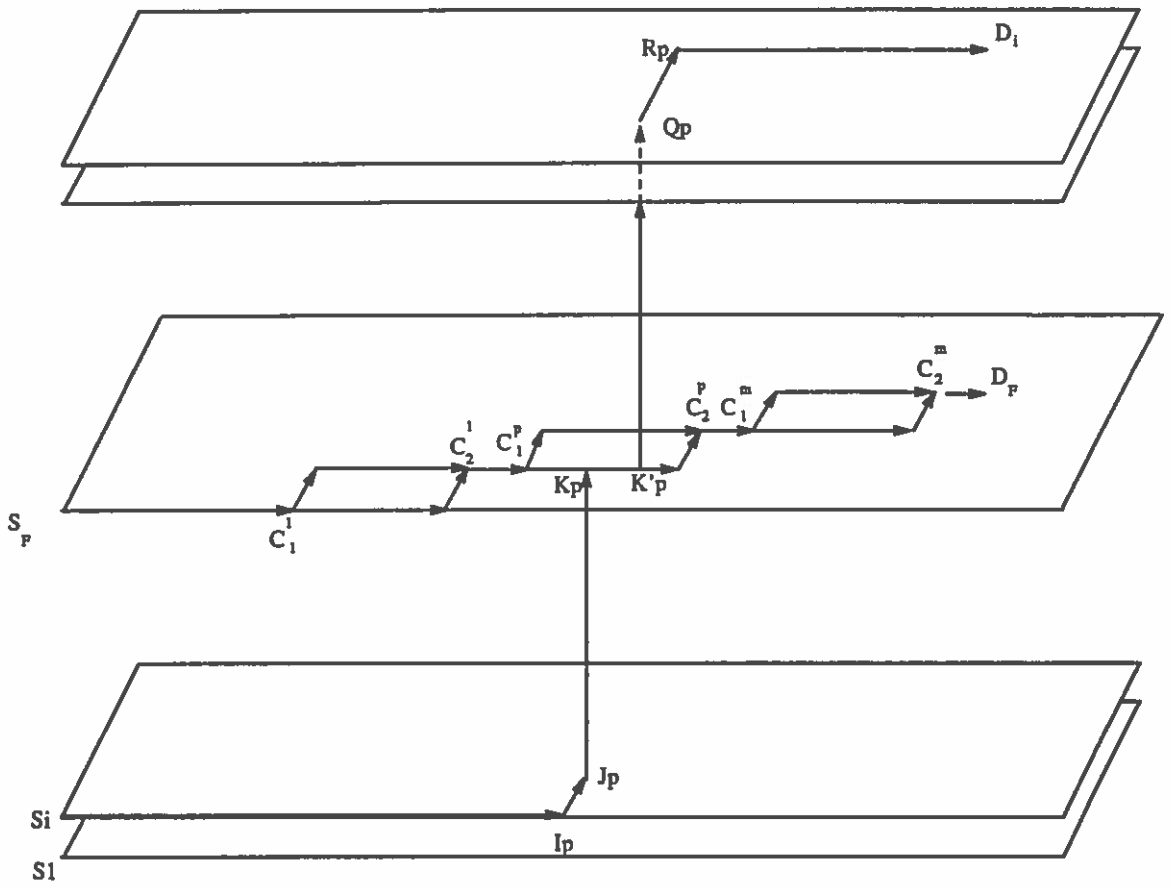
We claim that the 3-Sat problem has a true assignment iff R_P has a contention-free m-routing in A.

($\Rightarrow$) If the 3-Sat problem has a true assignment, the i-th clause (for any $i = 1, \ldots, k$) has at least one literal, say, x_p or $\bar{x_p}$ for the smallest index p which is true under the assignment. By the construction of A_1, there is a path in A_1 from S_i to D_i which passes a path segment from K_p to K'_p in rectangle p. We choose this path as the route for the message from S_i to D_i. Since we choose K_p and K'_p consistently: if x_p occurs, we choose lower nodes; if $\bar{x_p}$, we choose upper nodes, either upper or lower nodes in a rectangle are chosen, but not both. Therefore, we can choose the spare upper trail or lower trail as one of path segments for message from S_F to D_F. For other messages whose source and destination nodes form an edge not in A_1, we simply choose this edge as the route for the message. This routing is a contention-free m-routing.

(⇐) Supposed that R_P has a contention-free m-routing. First of all, notice that messages from S_i to D_i and message from S_F to D_F must be routed through edges in A_1 since for every edge (a, b) not in A_1, there is a message to be routed from a to b, which makes (a, b) the only possible route for a m-routing. This constrains the routes for the remaining messages to edges in A_1. For message from S_F to D_F to be unobstructed, only one trail (either lower part or upper part but not both) in any rectangle in plane $z = k + 1$ can be routed for messages from S_i to D_i. We construct a truth assignment as follows: for any $p = 1, \ldots, m$, if message from S_F to D_F is routed through upper trail in rectangle p, x_p is assigned *True*, otherwise, x_p is assigned *False*. Since there exists at least one unobstructed path P_i for message from S_i to D_i and P_i contains at least one edge in a rectangle, say, p, this means that clause i has an occurrence of literal x_p or $\bar{x_p}$. Furthermore, if P_i contains a lower edge in rectangle p, we know clause i has an occurrence of x_p, which has been assigned **True**. If P_i contains an upper edge in the rectangle, we know clause i has an occurrence of $\bar{x_p}$. However, in this case, x_p is assigned *False*. In both cases, the truth value of clause i is *True*. This concludes that the 3-Sat problem is satisfiable.

Since the above construction can be finished in polynomial time, we prove the theorem. □

The case for a two dimensional mesh is unknown. We conjecture that it is still NP-complete for the contention-free decision problem. Therefore, the Optimization Problem is NP-hard.



Let $W = 2\sum_{i=1}^{m} L_i$

$C_1^1 = (2, 2, k+1)$	$S_1 = (1, 1, 1)$	$D_1 = (2+W, m+3, k+2)$
$C_2^1 = (1+2L_1, 3, k+1)$	$S_2 = (1, 1, 2)$	$D_2 = (2+W, m+3, k+3)$
...		...
$C_2^m = (1+W, m+2, k+1)$	$S_k = (1, 1, k)$	$D_k = (2+W, m+3, 2k+1)$
$S_F = (1, 2, k+1)$		
$D_F = (2+W, m+2, k+1)$		

Figure 32: Illustration of the Proof of Theorem 1.

A Heuristic Algorithm

Based on the NP-completeness result in the previous section, it is therefore justified to design an efficient heuristic algorithm for the optimization problem. In this section, we present a simple and efficient heuristic algorithm $BLOCK$.

The idea behind heuristic $BLOCK$ is that, in an n dimensional mesh, given a message to be routed from source node $s = (s_1, \ldots, s_n)$ to destination node $d = (d_1, \ldots, d_n)$, we know that it is going to be routed in a shortest Manhattan path from s to d. This means that it can only pass through the *directed* edges in the rectangle $\{(x_1, \ldots, x_n) | s_i \leq x_i \leq d_i\}$. We call the rectangle an "affected rectangle". Here, the direction of an edge in the rectangle is determined by the direction from s to d as follows: if $s_i < d_i$, then for any edge $\{(x_1, \ldots, x_n), (y_1, \ldots, y_n)\}$ in the "affected rectangle", $y_i - x_i \in \{0, 1\}$.

We further associate a weight label for a directed edge in the mesh. The labeling scheme initially labels all edges as zero and then examines messages one by one. When a directed edge is in the affected rectangle of the message considered, its label is increased by one. These labels represent the potential contention resulting from routing of the messages. Figure 33 shows the labeling of a 6×5 mesh. We also observe that, for a message to be routed from s to d, the number of all shortest Manhattan paths can be calculated easily. This number is called *Freedom* for the message and is determined by the absolute differences of all components of s, d. Let $x_i = |s_i - d_i|, i = 1, \ldots, n$, we denote the freedom function as $F(x_1, \ldots, x_n)$. F can be calculated recursively as shown in Figure 34.

In particular, when $n = 2$, $F(x_1, x_2) = \begin{pmatrix} x_1 + x_2 \\ x_1 \end{pmatrix}$.

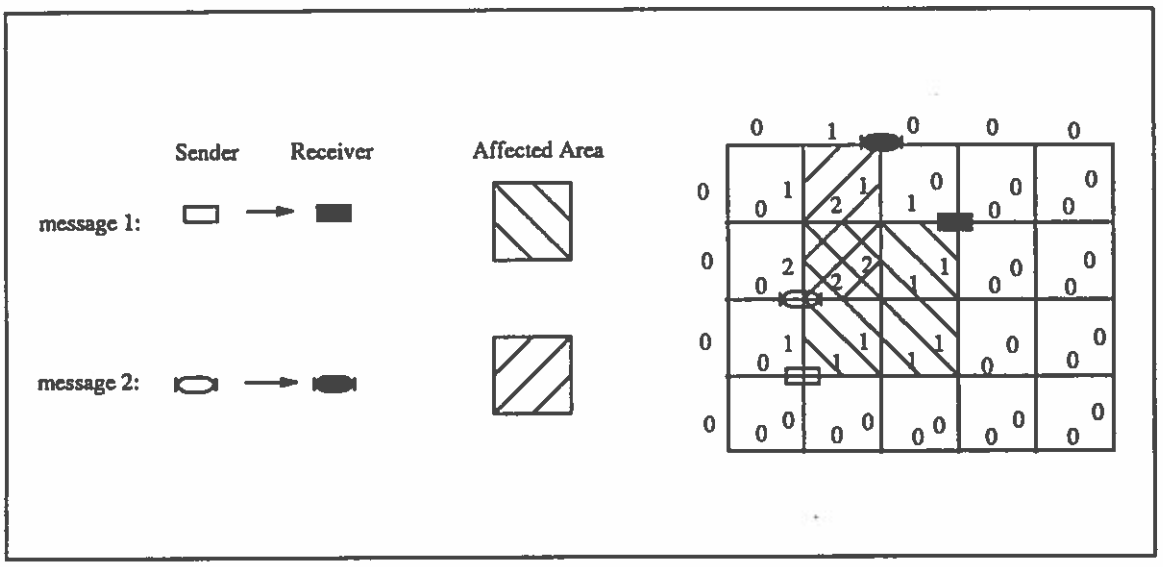


Figure 33: Illustration of the Labeling Scheme in $BLOCK$.

$$
\begin{aligned}
F(x_1,\ldots,x_n) &= F(x_1-1,\ldots,x_n)+\ldots+F(x_1,\ldots,x_n-1)\\
F(0,x_2,\ldots,x_n) &= F(0,x_2-1,\ldots,x_n)+\ldots+F(0,x_2,\ldots,x_n-1)\\
&\ldots\\
F(x_1,0,\ldots,0) &= 1\\
&\ldots\\
F(0,0,\ldots,0,x_n) &= 1
\end{aligned}
$$

Figure 34: Calculating Freedom Function.

For a message, the smaller its *Freedom* value, the fewer paths are eligible for its routing. To route messages on a labeled mesh, $BLOCK$ first sorts messages based on their *Freedom* values in an increasing order and then routes messages with lowest *Freedom* first. Routing an individual message is done by seeking a shortest Manhattan path such that maximum weight among all edges in the path is minimized. Such a path is called a shortest *minimum* path. We can use standard techniques for shortest paths such as Dijkstra's algorithm [2] to find such a path. Here, the distance of a path is defined as the maximum weight (label) of edges in the path. In fact, a more careful examination reveals that the subgraph formed by all shortest Manhattan paths for a message is an acyclic graph and hence, a more efficient shortest-path algorithm for acyclic graphs can be used to find such a *minimum* path.

After finding such a path P for a message m, labels of edges which are in the affected rectangle of message m but not in P are decreased by one. This is because after m is routed through P, m will not have any potential contention effect on other edges not in P. The new labels serve for finding shortest *minimum* paths for the next message.

The outline of the algorithm $BLOCK$ is shown in Figure 35. Figure 36 illustrates how $BLOCK$ works for a simple example.

The time complexity of $BLOCK$ can be analyzed as follows. Let A be an n dimensional mesh with N nodes. The number of edges it has is denoted as K. The number of messages is $|M|$. *Step 1* takes K time. The worst time for *Step 2* is $\mathcal{O}(|M|K)$. The time to calculate freedom for each message is also no more than $\mathcal{O}(|M|)N$ since we can compute $F(x_1, \ldots, x_n)$ in $x_1 x_2 \ldots x_n$ time steps based on the recursive relation in Figure 34. For a fixed-size $N_1 \times \ldots \times N_n$ mesh, we can even

```
/* Input: Set of Messages M, n dimensional mesh A */
/* Output: A m-routing */

/* initialization */
1: for   edge e ∈ A {label(e)=0;}
/* labeling */
2: for   message m ∈ M {
         for edge e in the affected rectangle of m {label(e)++;} }
/* Freedom calculation */
3: for   message m ∈ M { F(m)=calculate-freedom(m);}
/* sorting based on freedom F */
4: M=sort(M);
/* routing a message */
5: for   message m ∈ M {
         find a shortest minimum path P for m in A;
         for edge e in the affected rectangle of m but not in P {label(e) − −;} }
```

Figure 35: Outline of Heuristic BLOCK.

calculate the freedom function off-line for all values $F(x_1, \ldots, x_n), x_i \leq N_i$ by the recursive relation in Figure 34. The sorting step takes $|M|\log(|M|)$ steps. Finally, *Step 5* takes at most $|M|K$ time to complete since we can use the shortest path algorithm for acyclic graphs (see, page 203 in [84]) whose complexity is only $|E|$ where $|E|$ is the number of edges of the graph. This gives us the total complexity $\mathcal{O}(|M|(K + \log(|M|)))$. But since $K \leq nN$, the time complexity is $\mathcal{O}(|M|(nN + \log(|M|)))$.

Performance

Two test suites are used to evaluate $BLOCK$. In both tests, performance is evaluated with respect to maximum contention. The performance of $BLOCK$ is compared with that of the XY fixed-routing scheme.

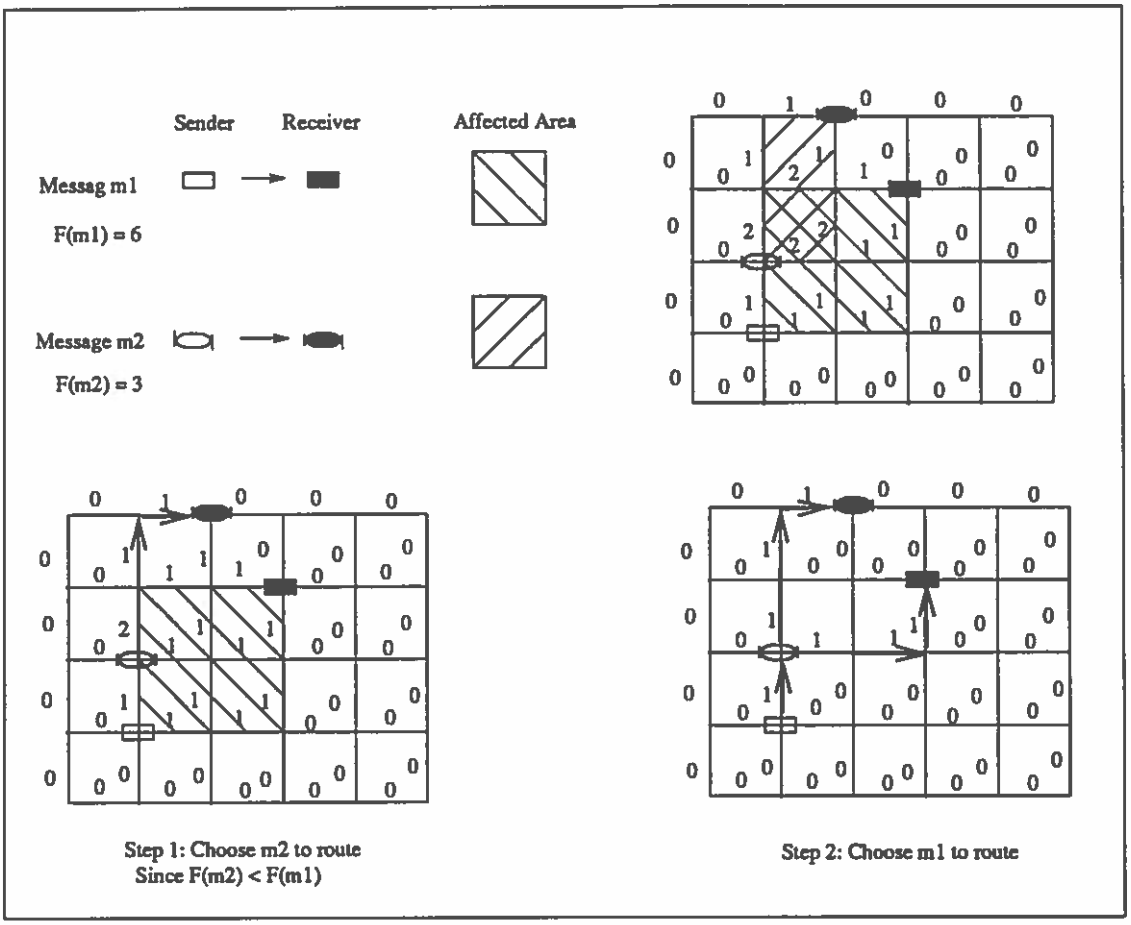


Figure 36: A Simple Example for the *BLOCK* Algorithm.

In the first test suite, messages are randomly distributed with a uniform distribution. Two independent uniform random number generators are used to generate source nodes and destination nodes respectively. The number of messages ranges from 10 to 500. The topologies used are 2-D 15×15 and 2-D 20×20 meshes. For a given number of messages, maximum contention is averaged over 100 runs of $BLOCK$. It is observed that the standard deviation of the mean maximum contention for all data point is less than 3%.

Figure 37 shows the maximum contentions of the XY-routing and the routing generated by $BLOCK$. Figure 38 shows the percentage improvement over XY-routing for maximum contention in the two meshes. It can be seen that as the number of messages grows, the percentage improvement drops. This is because when more and more messages are injected to the network, the network is more and more saturated, and it is harder for $BLOCK$ to reduce the contention.

The second test suite includes five benchmark applications representing a variety of task structures. The first is 16 node perfect broadcasting on a 2-D 4×4 mesh where processes exchange information to achieve a consensus [37]. The second is Gaussian elimination of a 32×32 matrix on a 2-D 8×4 mesh where each process is responsible for an entry of the matrix . The third one is the 15-body problem on a 2-D 5×3 mesh which was designed for the computation of planetary gravitational forces for the Caltech Cosmic Cube [5]. The fourth is an 18 node program called AVHTST on a 2-D 4×4 mesh which is used to determine cloud properties from satellite imagery data [104, 57] and finally, matrix transpose for various sizes. The task structures of all the above problems but the last are first assigned to the processors by a simple, greedy heuristic. For the matrix transpose problem, a 2-D mesh which has the same

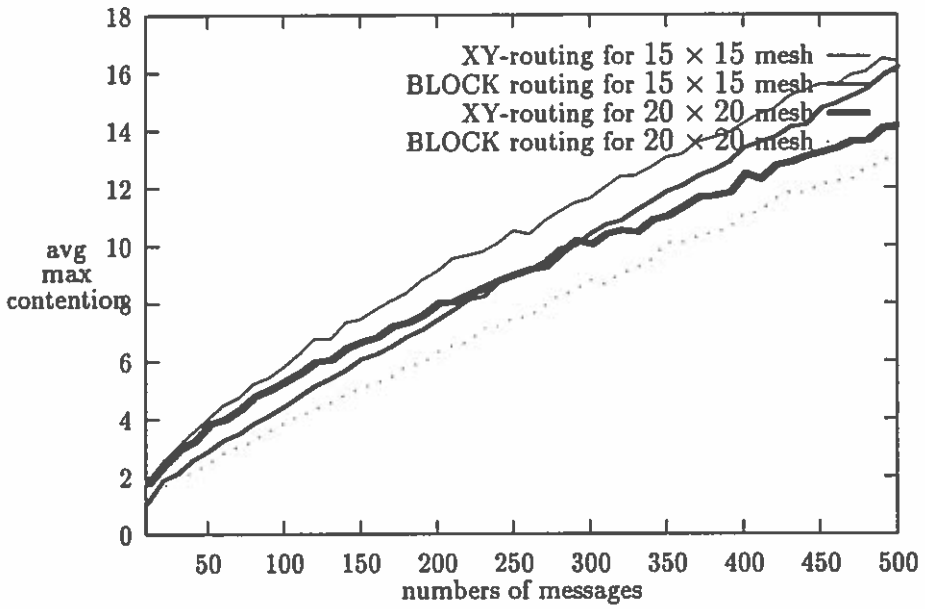


Figure 37: Average Maximum Contention for 2-D 15 × 15 and 20 × 20 Meshes under Uniform Message Distribution.

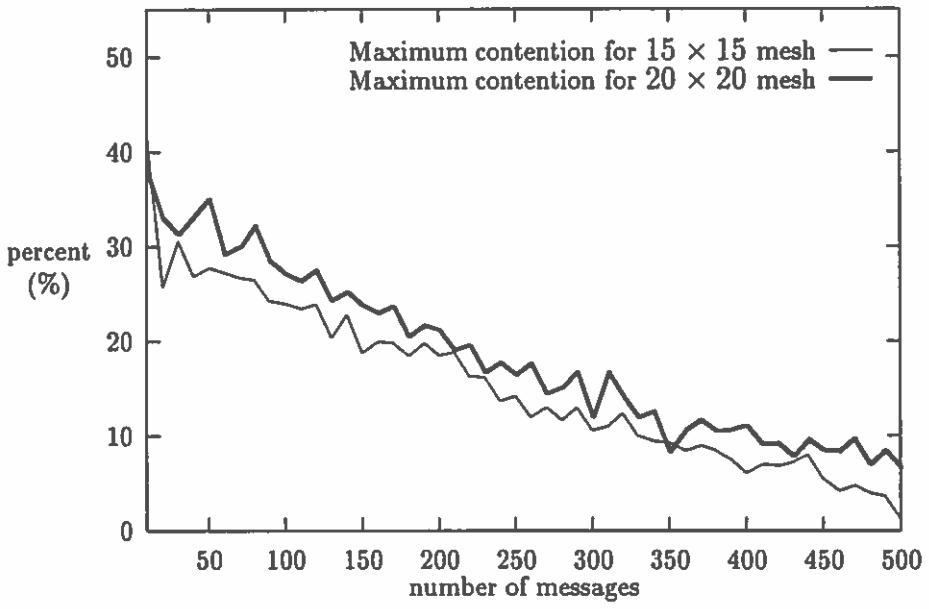


Figure 38: Percentage Improvement of Maximum Contention for 2-D 15 × 15 and 20 × 20 Meshes under Uniform Message Distribution.

size as the matrix is used, and a natural one-to-one processor assignment is used (i.e., the (i, j)-entry of the matrix is assigned to processor (i, j)).

Table 12 shows simulation data for the first four applications. In all cases, there is significant improvement with respect to maximum contention. Figure 39 shows the percentage improvement of maximum contention of the routing produced by $BLOCK$ over that of XY-routing for matrix transpose. The sizes of matrices range from 10×10 to 19×19. It can be seen that an improvement of approximately 40% is achieved in all experiments. It is also interesting to note that, in applications like Gaussian elimination and matrix transpose, message contention is very heavy. In fact, in the 32 Gaussian elimination benchmark, the maximum contention of XY-routing is 60 and, in 19×19 matrix transpose, the maximum contention of XY-routing is 29. However, $BLOCK$ is still able to reduce the maximum contention considerably (40% for Gaussian elimination and 20% for the matrix transpose). This is because in such applications, unlike in the case where messages are distributed *uniformly*, there exists communication locality for further improvement.

Table 12: Performance of $BLOCK$ for the Applications

Applications	Maximum Contention		
	XY-routing	$BLOCK$	Percentage
16 Perfect Broadcasting on 4×4 mesh	8	7	12.5%
32 Gaussian Elimination on 8×4 mesh	60	40	33.3%
15-body on 5×3 mesh	5	3	40.0%
18 node AVHTST on 4×4 mesh	7	6	14.3%

The performance of $BLOCK$ has been further studied by Duncan [33] where $BLOCK$ was compared with XY-routing and an adaptive routing scheme. Simulation was carried out to model runtime overhead to support the virtual channel capability.

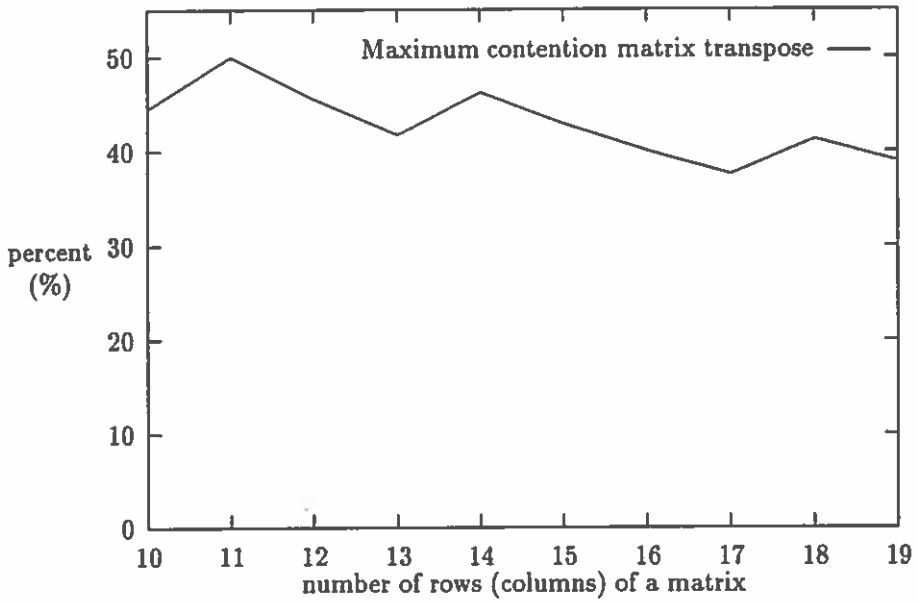


Figure 39: Percentage Improvement of Maximum Contention for the Matrix Transpose.

Based on the detailed simulation results for several benchmarks such as FFT, matrix multiplication and matrix transpose, Duncan concluded that for benchmarks which have intensive communication, *BLOCK* performs much better than the fixed XY routing and better than the adaptive scheme. For benchmarks which do not have intensive communication, however, *BLOCK* does not give performance advantage for the fixed-routing, which has very little runtime overhead.

Conclusions

We have presented an efficient heuristic for application-specific wormhole routing in a partitioned mesh-connected multiprocessor system. Such a heuristic can also be used for other routing schemes such as virtual cut-through and store-forward, and for the hypercube and other well-known topologies. The performance of the heuristic is studied for various benchmarks and uniformly distributed messages on the net-

work. For all performance suites, the heuristic achieves very good performance. Such an application-specific routing technique can be used in many architectures and is especially useful for real-time applications which require high performance.

The results presented here have several potential applications. First of all, the heuristic can be applied to the hypercube, torus and other interconnection structures. For example, it has also been pointed out in [74] that, to avoid deadlock, an n-dimensional binary cube can be partitioned into 2^n virtual networks (using the similar technique presented on page 26). All possible shortest paths are captured in these virtual networks. A similar "affected area" concept can be used to label communication channels in the cube. Furthermore, although our original motivation is for a partitioned mesh with wormhole routing, the heuristic can be applied to other routing schemes such as virtual cut-through or store-forward, provided that we are seeking a minimal routing.

The idea of generating application-specific routings can be potentially used in a parallelizing compiler. To use the heuristics, however, the message passing requirements must be known at compile time. We discuss this problem for different programming languages.

In a language where the communication structure can be explicitly declared by the programmer and a mapping is known at compile time, the message passing requirement can be extracted from the mapped communication structure directly. Languages with explicit communication structure declaration include Paragraph [27], Port Emsembles [48], and LaRCS [81] *.

The other languages whose program's communication structure can be known

*Strictly speaking, LaRCS is not a complete programming language, but it can be extended to be a programming language.

at compile time include VHDL [75] and ADA. In VHDL, for example, processes communicate with each other through signals. If a static mapping is performed, the communication structure can be deduced from the signals which are used as the communication channels for two processes. Furthermore, since the type of a signal has to be known after elaboration (which is part of the compiling process), the volume of the messages can be also known at compile time. Thus, for a parallel VHDL simulation, the application-specific routing techniques can be applied.

The second type of languages are those which support a shared data structure view. These include Kali [61] and Fortran90D [39]. Parallelism can be expressed in a shared-memory-like style and the programmer does not need to explicitly take care of the underlying communication. The compiler is responsible for the extraction of the communication structure, either at compile-time or at runtime. If the communication structure can be extracted at compile-time, the application-specific routing can be also generated at compile time and the routing information can be added to message passing commands.

Another interesting approach to extract the communication structure is by profiling. This approach has not drawn much attention.

Future research includes applying the technique to practical applications in practical machines, improving the heuristic and analyzing its theoretical behavior. Finally, we conjecture that even for 2-D meshes, the contention-free decision problem is still NP-complete. This seems to require a different approach than the one we presented for n-dimensional meshes for $n > 2$.

Part II

MAPPING TO SYSTOLIC ARRAYS

CHAPTER VI

SYSTOLIC ARRAY DESIGN

Many applications demand high performance. For such applications, hardware tailored to the application is designed to achieve the best performance. In electronic design, ASICs (Application Specific Integrated Circuits) have been widely used. As in mapping a parallel program onto a general purpose multicomputer, one of the major challenges in designing an application specific architecture is how to automate the design process and achieve high performance.

The problem of synthesizing a high performance application specific architecture from a high-level description of an application has long been studied. Recent successful commercial synthesis systems such as the Synopsys Design Compiler [53] indicate that synthesis techniques from RTL (Register Transfer Level) have become practical.

Synthesizing from high-level algorithmic description (behavioral level) of an application, however, is much more complicated. The complexity arises from both the high-level description (which can be as general as an arbitrary program) and the large design space of architectural choices. Many applications, however, have regular computation structures, which can often be exploited by designing regular architectures such as *systolic arrays*.

The second part of this dissertation is devoted to the problem of synthesizing systolic arrays from a special class of algorithms called regular iterative algorithms.

In this chapter, we first introduce the background on systolic array architectures, regular iterative algorithms and the classic techniques used in synthesizing systolic arrays from such algorithms. We then give an overview of Part II.

Systolic Arrays

Systolic arrays are special-purpose, massively parallel architectures designed for a single family of algorithms such as Matrix multiplication, LU decompositions [66]. The most important features in a systolic array are its space and time regularity. Such regularity makes systolic array processors amenable to VLSI implementation.

Intuitively, a pure systolic array has the following features [67, 98].

1. Consists of *identical* simple processors;
2. Processors are connected in a *topologically uniform* way;
3. All processors are governed by a global clock (i.e. a synchronous circuit). Data are rhythmically computed and passed;
4. Processors have simple functional units (such as a multiplier or an adder) which finish a computation in a fixed time period (for example, one clock cycle).

Systolic arrays are usually designed for a family of problems. For example, a *family* of arrays are designed for $N \times N$ matrix multiplication. The problem size here is the matrix order N. Such a family is said to be parametrized. Systolic arrays are scalable in the sense that larger problems can be solved simply by adding more processors. To ensure regularity and scalability, the interconnections and the processor structure of systolic arrays should be independent of the problem size.

The formal definition of a *pure* systolic array has been given by Rao and Kailath [98, 99]. We will use this definition throughout the thesis. Formally, a systolic array consists of a network of processors which are placed in a domain $\mathcal{D} \in Z^n$ (Z is the set of all integers) of integer points. It has the following properties.

1. If there is a communication channel from (to) the processor at location I to (from) the processor at location $(I + d)$, then
   (a) d is independent of the size of the problem,
   (b) for any $I \in \mathcal{D}$, there is such a connection. If $(I+d)$ is outside of the domain, then this is an output (input) channel. Notice that this connection also uniquely identifies the output (input) ports for a processor.
2. All processors repeatedly execute the same program (called iteration-unit) synchronously. The iteration-unit is independent of the size of the problem.

The first property ensures the spatial regularity and the second one ensures temporal regularity, and scalability.

Regular Iterative Algorithms

Designing a systolic array manually for a given problem is a difficult task. It is also hard to verify the correctness of the algorithm designed manually due to complicated data flow structures and timing relationships. Thus, it is important to automate this design process.

Early work on systolic arrays was based on researchers' ingenious design for specific systolic arrays such as Kung-Leiserson's arrays for matrix multiplication [67], Rote's arrays for the algebraic path problem, and Kung-Lo-Lewis' arrays for transitive

closure [69]. Moldovan [85], Quinton [93], Rao [98] and other researchers pioneered in automatic systolic array design, where the high-level description language is restricted to RIAs. Recently, there have been tremendous research efforts in synthesis of systolic arrays from a higher-level language called *Affine Recurrence Equation* [97, 96, 103, 119].

To be more precise, we are concerned with computations defined over an integral index domain, i.e. a subset of Z^n. A commonly used high-level specification language can be described as a system of *Recurrence Equations*. A recurrence equation is defined as follows.

$$C(\vec{I}) = g(\ldots C(f(\vec{I}))\ldots)$$

where $C(\vec{I})$ represents a computation at index point $\vec{I}$ for a certain *computation domain* $\mathcal{D}$ which is a subset of Z^n, $f(\vec{I}) : Z^n \rightarrow Z^n$ is a dependency function, $\ldots$ indicates other such possible arguments and g represents an *atomic* computation which can be executed within a clock cycle. A *system* of recurrence equations consists of finitely many (mutually recursive) such equations.

The computation domain $\mathcal{D}$ is usually a convex polyhedra represented as $\mathcal{D} = \{\vec{I} | A\vec{I} \geq \vec{b}, \vec{I} \in Z^n\}$. Furthermore, since systolic arrays are usually designed for a *family* of algorithms, we assume that the computations domains are a *family* of convex polyhedra.

By restricting the dependency function $f(\vec{I})$ in a recurrence equation, we can have:

- *Affine Recurrence Equation (ARE):* if $f(\vec{I}) = B\vec{I} - \vec{d}$ where B and $\vec{d}$ are constant matrix and vector respectively.

- *Uniform Recurrence Equation (URE):* if $f(\vec{I}) = \vec{I} - \vec{d}$ for some constant vector $\vec{d}$. In this case, $\vec{d}$ is called dependency vector.

The concept of URE was first introduced by Karp, Miller and Winograd [59] to study the combinatorial processes in solving differential equations by finite difference equations. Rao and Kailath [98, 99] used Regular Iterative Algorithms (RIAs) to describe the initial algorithms they used for the design of a systolic array. In fact, an RIA can be succinctly represented as a system of UREs. The concept of ARE was first introduced by Rajopadhye [97].

A classic method to synthesize systolic arrays from AREs is to first transform the AREs to UREs by so-called *Uniformization* technique which tries to transform linear dependencies to constant dependencies [97, 96, 20]. The techniques to synthesize systolic arrays from UREs are then used [98, 93]. Figure 40 illustrates systolic array design processe.

In this thesis, we are only concerned with synthesis from a single URE.

We say that $\vec{I} \in \mathcal{D}$ *depends directly* on $\vec{J} = \vec{I} - \vec{d} \in \mathcal{D}$ and denote it as $\vec{I} \rightarrow \vec{J}$. Intuitively, $\vec{I} \rightarrow \vec{J}$ means that $c(\vec{I})$ needs $c(\vec{J})$ as one of its arguments.

The following is the matrix multiplication example.

Example 6.1

$N \times N$ Matrix multiplication $Z = X \times Y$ can be described by a URE over a domain $\mathcal{D} = \{(i,j,k)^t | 1 \leq i \leq N, 1 \leq j \leq N, 1 \leq k \leq N\}$. The class of matrix multiplications is represented as a family of domains $\mathcal{D}$ for parameter N. The URE can be described as follows.

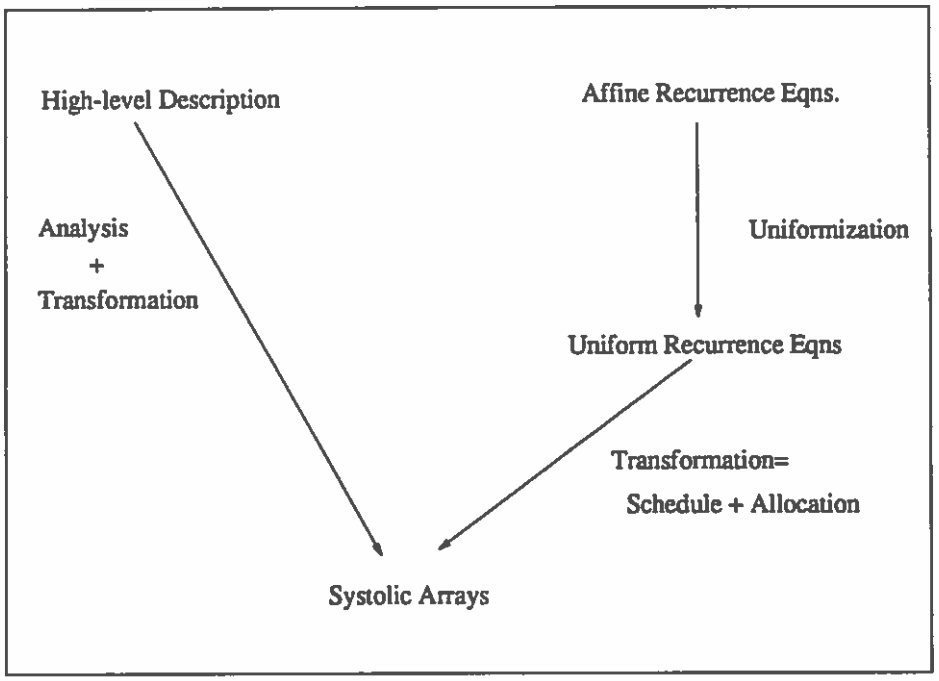


Figure 40: Illustration of Systolic Array Design Process.

for all $(i,j,k)^t \in \mathcal{D}$

$$\begin{cases} x(i,j,k) &= x(i,j-1,k) \\ y(i,j,k) &= y(i-1,j,k) \\ z(i,j,k) &= z(i,j,k-1) + x(i,j-1,k) \times y(i-1,j,k) \end{cases}$$

and the boundary conditions are

$$\begin{cases} x(i,0,k) &= X_{ik} \\ y(0,j,k) &= Y_{kj} \\ z(i,j,0) &= 0 \\ Z_{ij} &= z(i,j,N) \end{cases}$$

□

Mapping to Systolic Array Processors

A standard way to map an RIA to a systolic array is to use so called space-time transformations. A *schedule* (also called *the timing function*) is used to assign time steps to computations, based on the dependency constraints of the algorithm. An *allocation function* is used to allocate computations to physical processors.

A schedule S is a function over Z^n which maps a point $\vec{I} \in \mathcal{D}$ to a nonnegative number $S(\vec{I})$ with the condition that it obeys the dependencies in $\mathcal{U}$, i.e., if $\vec{I} \rightarrow \vec{J}$, then $S(\vec{I}) > S(\vec{J})$. Intuitively, S schedules computation $c(\vec{I})$ at piont $\vec{I}$ to be executed at time $S(\vec{I})$. The URE is called *computable* if there exists a schedule for it. A commonly used class of schedules is integer linear functions which are of form $S(\vec{I}) = \vec{\lambda}^t \vec{I} + \alpha$ where λ is an integer column vector and α is an integer. Such a schedule is called *linear schedule*. If a schedule S is of form $S(\vec{I}) = \lfloor \vec{s}^t \vec{I} + \alpha \rfloor$, it is called a

quasi-linear schedule. Linear and quasi-linear schedules are important because of their simplicity and the existence of linear programming methods to determine them optimally [59, 110].

The fastest execution for computation at each point in the domain is achieved when the *free schedule* [59] is used to schedule the computations. The *free schedule*, f, is a schedule defined as

$$f(\vec{I}) = \begin{cases} 0 & \text{if no } \vec{J} \in \mathcal{D} : \vec{I} \rightarrow \vec{J} \\ \max\{m | \vec{I} \xrightarrow{m} \vec{J}, \vec{J} \in \mathcal{D}\} + 1 & \text{otherwise} \end{cases}$$

Once a schedule is chosen, the next step is to allocate processes to processors. Such a step is usually realized by a function called an allocation function. A naive method to do that is to allocate a processor for a single computation point. This is, however, too wasteful of processor resources. In fact, the pipelining effect can be used such that computations which are scheduled to be processed at one processor are pipelined. A simple way to achieve this is to use a projection function to allocate computations. A commonly used allocation function is a single directional projection which can be represented either by an $(n-1) \times n$ integral matrix, A, or the projection vector $\vec{u}$. A linear schedule and a linear allocation function is called *a linear transformation.* They must satisfy the following two constraints.

- *Causality of the schedule S:* If, evaluating $c(\vec{I})$ needs value $c(\vec{I}-\vec{d})$, then $S(\vec{I}) > S(\vec{I}-\vec{d})$.
- *Non-conflict:* No two points are mapped to the same processor at the same time.

- *Dense array* The derived array must be dense i.e., every integral point in the processor space must be the image of an integral point in the index space of the problem.

Example 6.2

The following is an example for convolution product.

$$y(i) = \sum_{k=0}^{N} w(k)x(i-k)$$

It can be described as the following URE. *for* $i \geq 0$, **for** $0 \leq k \leq N$

$$\begin{aligned} Y(i,k) &= Y(i,k-1) + Y(i-1,k)X(i-1,k-1) \\ W(i,k) &= W(i-1,k) \\ X(i,k) &= X(i-1,k-1) \end{aligned}$$

Figure 41 shows a linear schedule, an allocation projection vector and the derived array. □

Overview of Part II

Although tremendous work has been done in designing optimal systolic arrays with respect to several criteria, several important problems remain open. Specifically, in this part of research, we solve the following problems.

- *Optimal linear schedule:* the open problem whether the optimal schedule of a URE can be obtained through a linear or quasi-linear function is answered. The

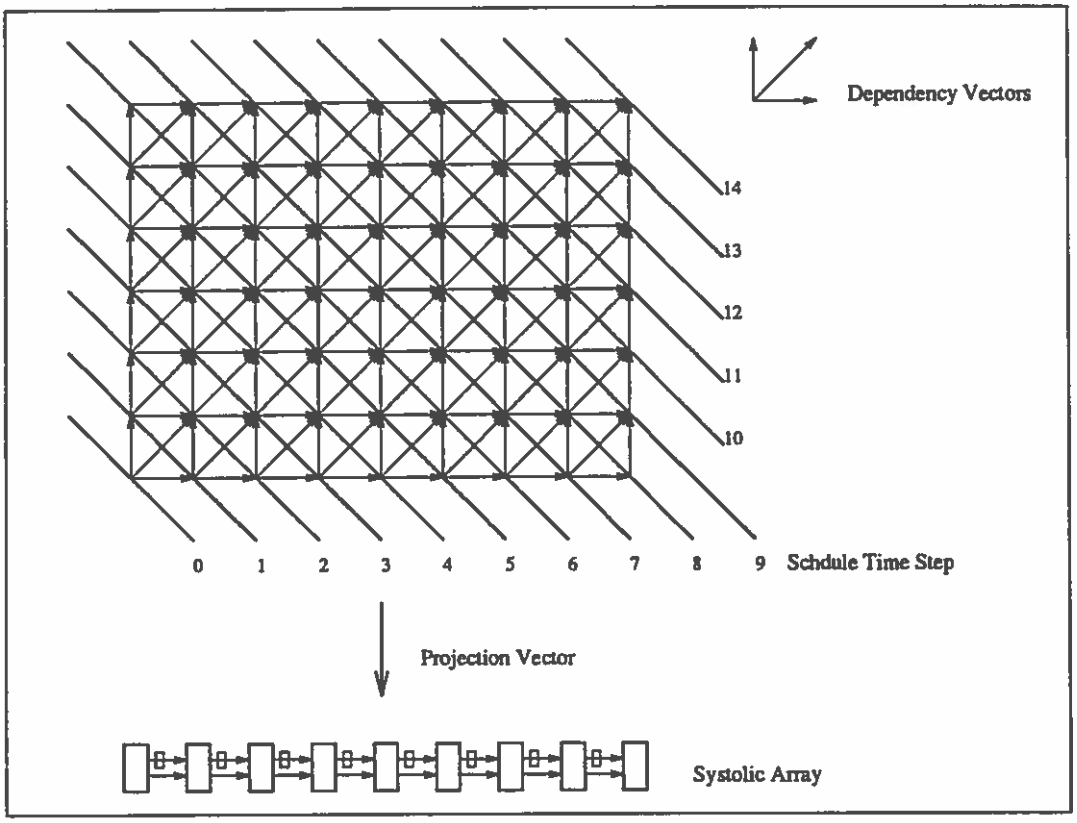


Figure 41: The Design of a Systolic Array for Convolution Product.

result justifies the use of linear schedules as the timing function in mapping a URE to a systolic array.

- *Linear allocation functions for systolic arrays with limited permissible interconnection:* the number of valid linear allocation functions which result in distinguished arrays is studied and it is found that this number is typically very small. A framework to design optimal systolic arrays based on this result is described.

- *Efficiency of a systolic array:* we propose a method to find quasi-linear allocation functions to derive systolic arrays which are almost 100% efficient. This result is further extended to any pure systolic array.

CHAPTER VII

OPTIMAL SCHEDULES FOR REGULAR ITERATIVE ALGORITHMS

Introduction

The problem of finding the optimal schedule (free schedule) for a URE was first attacked in a classic paper by Karp, Miller and Winograd [59]. They proved that the free schedule of an arbitrary URE over a specific index domain (the first orthant of the infinite multi-dimensional grid region, $\{\vec{I}|\vec{I} \geq 0\}$) is bounded within a constant to a rational linear function for computations not too "close" to the boundary. The problem of finding the optimal schedule for a URE over an arbitrary integral convex polyhedron and families of polyhedral domains, however, is still open.

Fortes and Parisi-Presicce studied linear schedules for 25 common algorithms of nested loops with constant dependence vectors [38]. They found that the difference between the optimal *linear* schedules and the optimal schedules is equal to one or zero for those nested loops. Extensive work on optimal *linear* schedules for UREs has been done by Shang and Fortes [110]. Efficient algorithms are designed to find optimal linear schedules for both the total completion time and the completion time of a specific computation. Recently, Darte, Khachiyan and Robert [32] proved the existence of a bound on the difference between the *total* computation time given by the *optimal linear schedule* and that given by the free schedule for sufficiently "fat" domains. Their result, however, does not reveal the exact nature of the free schedule for every computation point. Furthermore, they only considered a special class of

domains (namely, $\{\vec{I}|A\vec{I} \leq N\vec{b}\}$ where N is an integer as the domain parameter). In this chapter, we study the free schedule with respect to every computation point as well as the total computation time over a family of arbitrary integral polyhedra.

Notation and Problem Definition

Throughout this chapter, Z denotes the set of integers and Q, the set of rational numbers. The integral grid is Z^n. Vectors are column vectors and are denoted as $\vec{I}$. For a vector $\vec{I} = (i_1, \ldots, i_n)^t$, $|\vec{I}| = (|i_1|, \ldots, |i_n|)^t$ and $\|\vec{I}\|_\infty = \max\{|i_1|, \ldots, |i_n|\}$. E_n is an $n \times n$ identity matrix. $\vec{1} = (1, \ldots, 1)^t$. A sequence of integers $a_1, \ldots, a_k$ is called *semipositive* if they are nonnegative but not all of them are zeros. For a rational polyhedron $\mathcal{P}$, its *integer hull* $\mathcal{P}_I$ is defined as the convex hull of the integral vectors in $\mathcal{P}$. $\mathcal{P}$ is called an *integral polyhedron* (IP, see [107]), if $\mathcal{P}_I = \mathcal{P}$). Throughout this chapter, we assume that $\mathcal{P}$ is an IP.

Definition 7.1

Let A be an $m \times n$ integral matrix, an integral polyhedron IP is defined as $\mathcal{P} = \{\vec{I}|A\vec{I} \geq \vec{b}\}$j. Let S_b be a subset of Z^m, a family of IPs $\mathcal{F}(A, S_b)$ is $\{\mathcal{P}|\mathcal{P} = \{\vec{I}|A\vec{I} \geq \vec{b}\}$ for any $\vec{b} \in S_b$ and $\mathcal{P}$ is an $IP\}$. □

Intuitively, the "shape" of an IP is determined by its coefficient matrix A and $\mathcal{F}(A, S_b)$ is a collection of IPs which are of the same "shape". S_b is the range of the parameters. $\mathcal{P} = \{\vec{I}|A\vec{I} \geq \vec{b}\}$ can be decomposed as $\mathcal{P} = \mathcal{V} + \mathcal{C}$ where $\mathcal{V}$ is a polytope and $\mathcal{C} = \{\vec{y}|A\vec{y} \geq \vec{0}\}$ is the *characteristic* cone of $\mathcal{P}$ (cf. [107]). Notice that $\mathcal{C}$ is independent of $\vec{b}$.

From properties of an integral polyhedron, any integral point in an IP can be expressed as the sum of a positive combination of its rays and a convex combination

of its vertices. This corresponds to the following property.

Property 5 $\{I \in \mathcal{P}, I \text{ is integral}\} = \{e_1\vec{v}_1 + \ldots + e_q\vec{v}_q + f_1\vec{r}_1 + \ldots + f_p\vec{r}_p |\ e_i, f_j$ *nonnegative and* $e_1 + \ldots + e_p = 1\}$ *for some integral vectors* $\vec{v}_1, \ldots, \vec{v}_q$ *and* $\vec{r}_1, \ldots, \vec{r}_p$ *where* $\vec{r}_i$*s are independent of* $\vec{b}$*. (See [107], page 234, formula (19)).*

Since $\mathcal{P}$ is an IP, it is easy to see that $\mathcal{P} = \{e_1\vec{v}_1 + \ldots + e_q\vec{v}_q + f_1\vec{r}_1 + \ldots + f_p\vec{r}_p | e_i, f_j \geq 0$ and $e_1 + \ldots + e_q = 1\}$. Thus, *any* two $\mathcal{P}_1, \mathcal{P}_2 \in \mathcal{F}(A, S_b)$ have the same $\vec{r}_i$s. Throughout this chapter, we assume $\mathcal{P}$ has such a representation $\mathcal{P} = \{V\vec{e} + R\vec{f} | \vec{e}, \vec{f} \geq \vec{0}$ & $e_1 + \ldots + e_q = 1\}$ where $V = (\vec{v}_1 \ldots \vec{v}_q), R = (\vec{r}_1 \ldots \vec{r}_p)$.

Let us recall the definition of a URE.

Definition 7.2

Let c be a function (computation) on Z^n, A be a $m \times n$ integral matrix, a *U*niform *R*ecurrence *E*quation (**URE**) $\mathcal{U}$ over a domain $\mathcal{P}$ in a family of $\mathcal{F}(A, S_b)$ is

$$c(\vec{I}) \quad = \quad f(c(\vec{I} - \vec{d}_1), \ldots, c(\vec{I} - \vec{d}_k))$$

where $\vec{I} \in \mathcal{P}$, $\vec{d}_1, \ldots, \vec{d}_k$ are n-dimensional *integral* vectors and f is an arbitrary function. $\vec{d}_i$s are called *dependency vectors* and $D = (\vec{d}_1, \ldots, \vec{d}_k)$ is called *dependency matrix.* □

We say that $\vec{I} \in \mathcal{P}$ *depends directly* on $\vec{J} = \vec{I} - \vec{d}_i \in \mathcal{P}$ and denote it as $\vec{I} \rightarrow \vec{J}$. Intuitively, $\vec{I} \rightarrow \vec{J}$ means that $c(\vec{I})$ needs $c(\vec{J})$ as one of its arguments. Furthermore, if $\vec{I}_1 \rightarrow \vec{I}_2, \ldots, \vec{I}_n \rightarrow \vec{I}_{n+1}$ and $\vec{I}_1, \ldots, \vec{I}_{n+1} \in \mathcal{P}$, we denote $\vec{I}_1 \stackrel{n}{\rightarrow}_{\mathcal{P}} \vec{I}_{n+1}$. A *schedule* S is a function which maps an integral vector $\vec{I} \in \mathcal{P}$ to a positive integer such that if

$\vec{I} \rightarrow \vec{J}$ for any $\vec{J} \in \mathcal{P}$, then $S(\vec{I}) > S(\vec{J})$. Intuitively, a schedule is a function over $\mathcal{P}$ which schedules computation $c(\vec{I})$ at time $S(\vec{I})$ and obeys the dependencies. A URE is said to be *computable* in $\mathcal{F}(A, S_b)$ if there exists a schedule for any $\mathcal{P} \in \mathcal{F}(A, S_b)$. The shortest completion time for a computation is achieved when the *free schedule* f is used to schedule the computation.

If a schedule S is of form $S(\vec{I}) = \vec{\lambda}^t \vec{I} + \alpha$ where $\vec{\lambda}$ is a column integer vector, it is called a *linear schedule.* If a schedule S is of form $S(\vec{I}) = \lfloor \vec{s}^t \vec{I} + \alpha \rfloor$ where $\vec{s}$ is a column rational vector, it is called a *quasi-linear schedule.* Linear schedule is simple but sometimes a quasi-linear schedule is needed to schedule computation points at the earliest possible time.

Example 7.1

Consider the following URE where the computation domain $\mathcal{P} = \{(i,j)^t | 0 \leq i \leq 2N, 0 \leq j \leq 2N\}$ where N is a natural number representing the problem size parameter. The dependency matrix is as follows:

$$\begin{pmatrix} 2 & 2 \\ 0 & 1 \end{pmatrix}$$

Each point $(2k, j)$ can be scheduled at time k and each point $(2k+1, j)$ can be scheduled at time k too. In fact, the free schedule is $f(i,j) = \lfloor \frac{i}{2} \rfloor$. This is a quasi-linear schedule. □

Computability of a URE

Definition 7.3

A family of IPs is said to be *extensible* with respect to the dependency

vectors $\vec{d}_1, \ldots, \vec{d}_k$ of URE,$\mathcal{U}$, iff for any semipositive integers $l_1, \ldots, l_k$, there exists an IP $\mathcal{P}$ of the family such that it $\vec{I}_1 \xrightarrow{N}_{\mathcal{P}} \vec{I}_1 - \sum_{i=1}^{k} l_i \vec{d}_i$ for some integral point $\vec{I}_1 \in \mathcal{P}$ and some $N > 0$ (Note, N is not necessarily $\sum_{i=1}^{k} l_i$). □

Intuitively, for a family of IPs which is extensible with respect to dependency vectors, for an arbitrarily long dependency chain, we can always find an IP $\mathcal{P}$ in this family $\mathcal{P}$ contains two points one of which can be reached from the other through the dependency chain (not necessarily to be in $\mathcal{P}$).

For example, the family of IPs for matrix multiplication in Example 1 Chapter VI is extensible with respect to the dependencies. In the sequel, we assume that our family of domains are extensible with respect to all the dependency vectors of URE $\mathcal{U}$.

Intuitively, a URE is computable iff there is no infinite dependency chain.

Lemma 7.1

URE $\mathcal{U}$ is computable for a family $\mathcal{F}$ iff there is no semipositive integer sequence $a'_1, \ldots, a'_k$ such that $\sum_{i=1}^{k} a'_i \vec{d}_i = -\sum_{i=1}^{p} f'_i \vec{r}_i$ for some nonnegative integer f'_is. □

Proof

Suppose $\mathcal{U}$ is not computable over $\mathcal{P}$, then, there exists an infinite sequence of integral vector $\vec{I}_0(=\vec{I}), \vec{I}_1, \ldots, \vec{I}_n, \ldots$ in $\mathcal{P}$ such that $\vec{I}_i \rightarrow_{\mathcal{P}} \vec{I}_{i+1}$ (hence, for $m < n$, $\vec{I}_m - \vec{I}_n = \sum_{i=1}^{k} a_i \vec{d}_i$ for some semipositive integers a_i's). Based on the representation of $\mathcal{P}$, $\vec{I}_m = \sum_{i=1}^{q} e_i^m \vec{v}_i + \sum_{i=1}^{p} f_i^m \vec{r}_i$ for some nonnegative integers e_i^m's and f_i^m's and $= \sum_{i=1}^{q} e_i^m = 1$. Since

there are only finitely many (q) different values of e_i^m's (when one e_i^m is one, all of the rest must be zero), there exists an infinite subsequence of $\vec{I}_1, \ldots, \vec{I}_n, \ldots$ such that their e_i^m's are repetitive. Among this subsequence, It is always possible to further choose an infinite subsequence whose f_i^m are *nondecreasing* for each $i = 1, \ldots, p$. Let $\vec{I}_m$ and $\vec{I}_n$ be two points in such a subsequence ($m < n$), we have $\vec{I}_m - \vec{I}_n = \sum_{i=1}^{k} a_i' \vec{d}_i = -\sum_{i=1}^{p} f_i'$ for some semipositive integers a_i' and nonnegative integers f_i'.

Conversely, if there is a semipositive integer sequence $a_1, \ldots, a_k$ such that $\sum_{i=1}^{k} a_i \vec{d}_i = -\sum_{i=1}^{p} f_i \vec{r}_i$ for some nonnegative integer f_i's. Since $\mathcal{F}$ is extensible with respect to $\vec{d}_i$'s, there exists a $\mathcal{P} \in \mathcal{F}$ and an integral $\vec{I} \in \mathcal{P}$ such that $\vec{I}_0 (= \vec{I}) \rightarrow_{\mathcal{P}} \vec{I}_1 \rightarrow_{\mathcal{P}} \ldots, \vec{I}_N (= \vec{I} - \sum_{i=1}^{k} a_i \vec{d}_i)$. But $\vec{I}_i + \sum_{i=1}^{p} f_i \vec{r}_i \in \mathcal{P}$ (since $\vec{I}_i \in \mathcal{P}$ and recall $\mathcal{P}$'s representation), we have an infinite dependency chain $\vec{I}_0 \rightarrow_{\mathcal{P}} \vec{I}_1 \rightarrow_{\mathcal{P}} \ldots \vec{I}_N \rightarrow_{\mathcal{P}} \vec{I}_1 + \sum_{i=1}^{p} f_i \vec{r}_i \ldots$. Thus, $\mathcal{U}$ is not computable over $\mathcal{P}$. □

The above result is a generalization of a classic result obtained by Karp et. al. [59] where a similar condition is given for a URE over a specific domain (the first orthant of the gird).

Theorem 7.1

$\mathcal{U}$ is computable iff there exists an n-dimensional vector s such that

$$\begin{aligned} \vec{s}^t \vec{d}_i &\geq 1, i = 1, \ldots, k \\ \vec{s}^t \vec{r}_i &\geq 0, i = 1, \ldots, p \end{aligned} \qquad (1)$$

□

Proof

From Lemma 7.1, $\mathcal{U}$ is computable iff the linear programming problem (LP) $\max\{\sum_{i=1}^{k} a_i | \sum_{i=1}^{k} a_i\vec{d}_i + \sum_{i=1}^{p} f_i\vec{r}_i = 0, a_i, f_j \geq 0\}$ has a finite optimal value 0, iff its dual problem $\min\{0 | \vec{s}^t\vec{d}_i \geq 1, \vec{s}^t\vec{r}_i \geq 0\}$ has a feasible solution. □

Corollary 1 *$\mathcal{U}$ is computable iff there exists a quasi-linear schedule for each $\mathcal{P}$.*

Proof

Because there are only finitely many $\vec{v}_i$ in an IP $\mathcal{P}$ $(i = 1, \ldots, q)$, for any $\vec{s}$, we can always find a constant α such that $\vec{s}^t\vec{v}_i + \alpha \geq 0$ for all $\vec{v}_i$ in an IP $\mathcal{P}$. In particular, let $\vec{s}$ satisfy condition (1),It is easy to prove that $L(\vec{I}) = \lfloor \vec{s}^t\vec{I} + \alpha \rfloor$ is a valid schedule for $\mathcal{U}$ on $\mathcal{P}$ (simply check $L(\vec{I}) \geq 0$ and $L(\vec{I}_1) > L(\vec{I}_2)$ if $\vec{I}_1 \rightarrow \vec{I}_2$). □

Condition (1) is similar to a result given by Quinton [93]. When the family consists of only bounded (finite) IPs (i.e., $p = 0$), this is called "separating hyperplane" ([59, 93]): $\vec{s}$ can be thought of as the norm of a hyperplane which separates dependency vectors from the first orthant. In the bounded convex polyhedron case, Condition (1) has been used as part of the definition for a *quasi-linear schedule* [32, 110]. Furthermore, Darte et al. [32] assumes that the URE admits a quasi-linear schedule as their premise in their paper on the optimality of linear schedule.

All the subsequent results in this Chapter are equally valid even if the family of domains is not extensible but if all domains admit a quasi-linear schedule or satisfies Condition (1).

The Free Schedule

We first consider the following two dual *rational* linear programming problems for an *integral* vector $\vec{I} \in \mathcal{P}$.

$$\begin{cases} m_1(\vec{I}) = \max \sum_{i=1}^{k} u_i \\ \text{subject to} \\ 1)\ u_i \geq 0, i = 1, \ldots, k \\ 2)\ \vec{I} - \sum_{i=1}^{k} u_i \vec{d_i} \in \mathcal{P} \end{cases} \quad \text{(I)} \qquad \begin{cases} m_2(\vec{I}) = \min \vec{\lambda}^t (A\vec{I} - \vec{b}) \\ \text{subject to} \\ 1)\ \lambda_i \geq 0, i = 1, \ldots, k \\ 2)\ \vec{\lambda}^t A\vec{d_i} \geq 1, i = 1, \ldots, k \end{cases} \quad \text{(II)}$$

Lemma 7.2

If URE $\mathcal{U}$ satisfies Condition (1), then both (I) and (II) have a common, finite optimal value $m(\vec{I})$ for an integral vector $\vec{I} \in \mathcal{P}$. □

Proof

Based on the representation of $\mathcal{P}$, the feasible region of (I) can be rewritten as follows.

$$\begin{cases} m_1(\vec{I}) = \max \sum_{i=1}^{k} u_i \\ \text{subject to} \\ 1)\ \vec{u}, \vec{e}, \vec{f} \geq \vec{0} \\ 2)\ \vec{I} - D\vec{u} - V\vec{e} - R\vec{f} = \vec{0} \\ 3)\ \vec{1}^t \vec{e} = 1 \end{cases}$$

Its dual problem is

$$\begin{cases} \min\{(\vec{s}^t \ \ \alpha)\begin{pmatrix} \vec{I} \\ 1 \end{pmatrix} \\ \text{subject to} \\ 1)\ \vec{s}^t D \geq \vec{1}_k^t \\ 2)\ \vec{s}^t R \geq \vec{0}_p^t \\ 3)\ \vec{s}^t V + \vec{\alpha}_q^t \geq \vec{0}_q^t \end{cases} \qquad \text{(III)}$$

where α is a scalar variable, $\vec{\alpha}_q^t$ represents a q-dimensional row vector whose components are α, and $\vec{1}_k^t$ ($\vec{0}_{p+q}^t$) is a k-dimensional ($(p+q)$-dimensional) vector with all 1 (0) components. Since Condition (1) holds, the feasible region of (III) is nonempty (we can always choose α to satisfy $\vec{s}^t\vec{v}_i + \alpha \geq 0$ for all $i = 1 \ldots, q$). Hence (III) has a feasible solution. But the feasible region of (I) is not empty since $\vec{u} = \vec{0}$ is a feasible solution. Hence, by duality theorem, (I) has a finite maximum. Therefore, its dual problem (II) must have a finite minimum too. Thus, $m_1(\vec{I}) = m_2(\vec{I})$ and they are denoted as $m(\vec{I})$. □

Notice that the feasible region of (II) is independent of $\vec{b}$. The objective function of (II) is a linear function of $\vec{I}$ and $\vec{b}$. Based on a property in parametric linear programming (see, for example, [87] pp. 15), we can show that $m(I)$ is a piece-wise linear function of I and b. Furthermore, for a fixed $\mathcal{P}$ (i.e., b is fixed), we know that $m(I)$ is also a piece-wise linear function of I (a projection of a piecewise linear function is still a piecewise linear function).

Lemma 7.3

The integer function $m'(\vec{I}) = \lfloor m(\vec{I}) \rfloor$ is a valid schedule for URE $\mathcal{U}$ in $\mathcal{P}$. □

Proof

Let $\vec{I}_1, \vec{I}_2 \in \mathcal{P}$ and $\vec{I}_1 \xrightarrow{N}_{\mathcal{P}} \vec{I}_2$ for some $N = \sum_{i=1}^{k} u_i > 0$ where u_i's are nonnegative integers for $i = 1, \ldots, k$ $\vec{I}_2 = \vec{I}_1 - \sum_{i=1}^{k} u_i \vec{d}_i$. Let $D_{\vec{I}_2} = \{\vec{u}' = (u'_1, \ldots, u'_k) | \vec{I}_2 - \sum_{i=1}^{k} u'_i \vec{d}_i \in \mathcal{P}\}$. For any $\vec{u}' \in D_{\vec{I}_2}$, we have $\vec{I}_1 - \sum_{i=1}^{k} (u_i + u'_i) d_i \in \mathcal{P}$. Thus, $m(\vec{I}_1) \geq \max_{\vec{u}' \in D_{\vec{I}_2}} \sum_{i=1}^{k} (u_i + u'_i) = \max_{\vec{u}' \in D_{\vec{I}_2}} \sum_{i=1}^{k} (u'_i) + N$. But $m(\vec{I}_2) = \max_{u' \in D_{\vec{I}_2}} \sum_{i=1}^{k} u'_i$ and since $N > 0$, we conclude $\lfloor m(\vec{I}_1) \rfloor \geq \lfloor m(\vec{I}_2) + N \rfloor > \lfloor m(\vec{I}_2) \rfloor$. □

We call $m'(\vec{I})$ a piecewise quasi-linear schedule. One may conjecture that $m'(\vec{I})$ is the free schedule for the URE. However, this is not correct. The main reason is that for $\vec{I}_1, \vec{I}_2 \in \mathcal{D}$ such that $\vec{I}_2 = \vec{I}_1 - \sum_{i=1}^{k} u_i \vec{d}_i$, it is not necessarily true that $\vec{I}_1 \xrightarrow{N}_{\mathcal{P}} \vec{I}_2$ for $N = \sum_{i=1}^{k} u_i$. This is because the dependencies from $\vec{I}_1$ to $\vec{I}_2$ may first go out of $\mathcal{P}$ and then go back into $\mathcal{P}$.

However, we are able to prove that for computations which are not too "close" to the boundary, the difference between $m'(\vec{I})$ and $f(\vec{I})$ (f, the free schedule) is bounded within a constant which is independent of $\vec{b}$. The method used is similar to that in [59]. We first define another polyhedron $\mathcal{P}_s$ for computation points which are not too "close" to the boundary of $\mathcal{P}$ as $\mathcal{P}_s = \{I | A\vec{I} \geq \vec{b} + \vec{\pi}\}$ where

$$\vec{\pi} = \sum_{i=1}^{k} |A\vec{d}_i|.$$

Since $\vec{\pi} \geq \vec{0}$, $\mathcal{P}_s \subseteq \mathcal{P}$. We assume that $\mathcal{P}_s$ is not empty. We first prove the following lemma.

Lemma 7.4

If $\vec{I}_1, \vec{I}_2 = \vec{I}_1 - \sum_{i=1}^{k} u_i \vec{d}_i \in \mathcal{P}_s$ for some semipositive integers $u_1, \ldots, u_k$, then $\vec{I}_1 \xrightarrow{N}_{\mathcal{P}} \vec{I}_2$ where $N = \sum_{i=1}^{k} u_i$. □

Proof

We prove the following claims successively.

Claim 1: If $\vec{\delta} \in \mathcal{P}_s$ and $\theta_i \in [0,1], i = 1, \ldots, k$, then $\vec{r} = \vec{\delta} + \sum_{i=1}^{k} \theta_i \vec{d}_i \in \mathcal{P}$.

Proof of Claim 1: Simply check

$$\begin{aligned} A\vec{r} &= A\vec{\delta} + \sum_{i=1}^{k} \theta_i A\vec{d}_i = A\vec{\delta} - \vec{\pi} + (\sum_{i=1}^{k} \theta_i A\vec{d}_i + \vec{\pi}) \\ &= A\vec{\delta} - \vec{\pi} + (\sum_{i=1}^{k} \theta_i A\vec{d}_i + \sum_{i=1}^{k} |A\vec{d}_i|) = A\vec{\delta} - \vec{\pi} + \sum_{i=1}^{k} (\theta_i A\vec{d}_i + |A\vec{d}_i|) \\ &\geq A\vec{\delta} - \vec{\pi} \geq \vec{b} \end{aligned}$$

Claim 2: Let $\vec{r} = \vec{\delta} + \sum_{i=1}^{k} \theta_i$ be an integral vector for some $\delta \in \mathcal{P}_s$ for some $\theta_i \in [0,1], i = 1, \ldots, k$ and let $\vec{r} - \sum_{i=1}^{k} \alpha_i \vec{d}_i = \vec{r}'$ where $\alpha_i \in \{0,1\}$, then $\vec{r} \xrightarrow{M}_{\mathcal{P}} \vec{r}'$ where $M = \sum_{i=1}^{k} \alpha_i$.

Proof of Claim 2: Without loss of generality, we assume that $\alpha_i = 1, i = 1, \ldots, L$ and $\alpha_i = 0, i = L+1, \ldots, k$ for some $L \leq k$. Let $\vec{r}'_j = \vec{r} - \sum_{i=1}^{j} \vec{d}_i, j = 0, 1, \ldots, L$ (hence, $\vec{r} = \vec{r}'_0$ and $\vec{r}' = \vec{r}'_L$), we have

$$\begin{aligned} A\vec{r}'_j &= A\vec{r} - \sum_{i=1}^{j} A\vec{d}_i = A(\vec{\delta} + \sum_{i=1}^{k} \theta_i \vec{d}_i) - \sum_{i=1}^{j} A\vec{d}_i \\ &= A\vec{\delta} + \sum_{i=1}^{k} \theta_i A\vec{d}_i - \sum_{i=1}^{j} A\vec{d}_i = A\vec{\delta} + \sum_{i=1}^{j} (\theta_i - 1) A\vec{d}_i + \sum_{i=j+1}^{k} \theta_i A\vec{d}_i \\ &= A\vec{\delta} - \vec{\pi} + \sum_{i=1}^{j} (|A\vec{d}_i| + (\theta_i - 1)(A\vec{d}_i)) + \sum_{i=j+1}^{k} (|A\vec{d}_i| + \theta_i A\vec{d}_i) \end{aligned}$$

$$\geq \quad A\vec{\delta} - \vec{\pi} \geq \vec{b}$$

Hence, $\vec{r}'_j \in \mathcal{P}$. Therefore, $\vec{r} = \vec{r}'_0 \rightarrow_{\mathcal{P}} \vec{r}'_1 \rightarrow_{\mathcal{P}} \vec{r}'_2 \ldots \rightarrow_{\mathcal{P}} \vec{r}'_L = \vec{r}'$, we prove the claim.

Proof of Lemma 7.4: Let, for $c = 0, 1, \ldots, N$,

$$\vec{r}_c = \vec{I}_1 - \sum_{i=1}^{k} \lfloor \frac{cu_i}{N} \rfloor \vec{d}_i$$

$\vec{r}_0 = \vec{I}_1, \vec{r}_N = \vec{I}_2$. Rewrite $\vec{r}_c$ as follows

$$\vec{r}_c = \vec{I}_1 - \frac{c}{N} \sum_{i=1}^{k} u_i \vec{d}_i + \sum_{i=1}^{k} (\frac{cu_i}{N} - \lfloor \frac{cu_i}{N} \rfloor) \vec{d}_i$$

Since $\mathcal{P}_s$ is a convex polyhedron,

$$\vec{I}_1 - \frac{c}{N} \sum_{i=1}^{k} u_i \vec{d}_i = (1 - \frac{c}{N}) \vec{I}_1 + \frac{c}{N} \vec{I}_2 \in \mathcal{P}_s.$$

Based on *Claim 1*, $\vec{r}_c \in \mathcal{P}, c = 0, 1, \ldots, N$. Furthermore, notice that

$$\vec{r}_{c+1} = \vec{r}_c - \sum_{i=1}^{k} (\lfloor \frac{(c+1)u_i}{N} \rfloor - \lfloor \frac{cu_i}{N} \rfloor) \vec{d}_i,$$

and

$$\lfloor \frac{(c+1)u_i}{N} \rfloor - \lfloor \frac{cu_i}{N} \rfloor \in \{0, 1\}.$$

Based on *Claim 2*, $\vec{r}_c \stackrel{L_c}{\rightarrow_{\mathcal{P}}} \vec{r}_{c+1}$ where

$$L_c = \sum_{i=1}^{k} (\lfloor \frac{(c+1)u_i}{N} \rfloor - \lfloor \frac{cu_i}{N} \rfloor).$$

But $\sum_{c=0}^{N-1} L_c = N$, hence we prove the lemma. □

Theorem 7.2

For a family of IPs $\mathcal{F}(A, S_b)$, there exists a constant C such that for every $\mathcal{P} \in \mathcal{F}(A, S_b)$, $m'(\vec{I}) - f(\vec{I}) \leq C$ for any integer vector $I \in \mathcal{P}_s$. □

Proof

Consider the two (dual) rational linear programming problems (I) and (II), but now restricted to $\mathcal{P}_s$ rather than $\mathcal{P}$. These are formulated as follows. For any integral vector $I \in \mathcal{P}_s$.

$$\begin{cases} M_1(\vec{I}) = \max \sum_{i=1}^{k} u_i \\ \text{subject to} \\ 1)\ u_i \in Q \text{ and } u_i \geq 0, i = 1, \ldots, k \\ 2)\ I - \sum_{i=1}^{k} u_i \vec{d_i} \in \mathcal{P}_s \end{cases} \qquad \text{(I')}$$

$$\begin{cases} M_2(\vec{I}) = \min \lambda^t (A\vec{I} - (\vec{b} + \vec{\pi})) \\ \text{subject to} \\ 1)\ \lambda_i \geq 0, i = 1, \ldots, k \\ 2)\ \vec{\lambda}^t A \vec{d_i} \geq 1, i = 1, \ldots, k \end{cases} \qquad \text{(II')}$$

$M_1(\vec{I})$ is finite since $u_i = 0$ is a feasible solution and $M_1(\vec{I}) \leq m(\vec{I})$ (since $\mathcal{P}_s \subseteq \mathcal{P}$). Thus, both (I') and (II') have a common optimal value $M(\vec{I}) = M_1(\vec{I}) = M_2(\vec{I})$. Let the optimal solution of the *integer* linear programming by restricting u_is to be integers in (I') be $M'(\vec{I})$, based on a result in integer linear programming(see, page 239-240, theorem 17.2, in [107]), there exists a constant C_1 which is *independent* of $\vec{b}$ such that

$0 \leq M(\vec{I}) - M'(\vec{I}) \leq C_1$. Now, consider the difference between the free schedule $f(\vec{I})$ of $\vec{I}$ in $\mathcal{P}_s$ and $M'(\vec{I})$. Since $M'(\vec{I}) = \sum_i^k u_i$ (say $= N$) for some integer $u_i \geq 0$ such that $\vec{I}' = \vec{I} - \sum_i^k u_i \vec{d}_i \in \mathcal{P}_s$, from Lemma 7.4, we have $\vec{I} \xrightarrow{N}_{\mathcal{P}} \vec{I}'$. Thus, $f(\vec{I}) \geq N(= M'(\vec{I}))$. Hence,

$$m'(\vec{I}) - f(\vec{I}) \leq m'(\vec{I}) - M'(\vec{I}) \leq m(\vec{I}) - M'(\vec{I}) \leq m(\vec{I}) - M(\vec{I}) + C_1$$

Notice that (II) and (II') have optimal solutions and both of their feasible regions are the same (denoted as $\Delta = \{\vec{\lambda} | \vec{\lambda}^t A\vec{d}_i \geq 1, i = 1, \ldots, k, \vec{\lambda} \geq \vec{0}\}$). Let E_Δ be the (finite) set of extreme points of the convex polyhedra Δ. There exists $\vec{\lambda}' \in \Delta$ such that

$$M(\vec{I}) = \vec{\lambda}'^t(A\vec{I} - (\vec{b} + \vec{\pi})).$$

But

$$m(\vec{I}) \leq \vec{\lambda}'^t(A\vec{I} - \vec{b}),$$

we have

$$m(\vec{I}) - M(\vec{I}) \leq \vec{\lambda}'^t\vec{\pi} \leq \max_{\vec{\lambda} \in E_\Delta} \vec{\lambda}^t\vec{\pi}$$

Hence, letting $C = C_1 + \max_{\vec{\lambda} \in E_\Delta} \vec{\lambda}^t\vec{\pi}$, we prove the theorem. □

Karp et al. [59] show an example where for some points "close" to the boundary, the difference between $m(\vec{I})$ and $f(\vec{I})$ are not bounded to a constant for the first orthant of n-dimensional grid. By slightly modifying their example, we can also show that even for a family of bounded IPs, difference between $m(\vec{I})$ and $f(\vec{I})$ may still be unbounded. This is given in the following example.

Example 7.2

Consider a URE with three dependency vectors $\vec{d}_1 = (-1,1,1)^t$, $\vec{d}_2 = (1,-1,1)^t$ and $\vec{d}_3 = (0,0,2)$ over a family $\mathcal{F}(A, S_b)$ of $2N \times 2N \times 2N$ cubes (for $N \geq 1$). For $\vec{I} = (1,1,2N)^t$, $\vec{I} - (N-1)\vec{d}_1 - (N-1)\vec{d}_2 = (1,1,2)^t$. $m(\vec{I}) = 2N-2$. But $f(\vec{I}) = N-1$. Thus $m(\vec{I}) - f(\vec{I}) = N-1$ which can not be bounded to a constant independent of the size of the cube. □

The Optimal Schedule for The Last Computation

In many applications, one often desires a schedule that minimizes the completion time of the *whole* computation. This problem is meaningful only if the domain $\mathcal{P}$ is bounded (i.e., a polytope). Thus, in this section, $\mathcal{P}$ is assumed to be an integral polytope. A computation $c(\vec{I})$ is called a last computation in domain $\mathcal{P}$ if there is no other point $\vec{I}' \in \mathcal{P}$ such that $\vec{I}' \xrightarrow{N}_{\mathcal{P}} \vec{I}$ for some $N > 0$. It is possible that there are more than one last computations. A schedule which minimizes the completion time of the whole computation is one which schedules all the last computations at the time given by the free schedule. In this section, we show that if URE $\mathcal{U}$ has only a single last computation $c(\vec{I}_l)$, then there exists a *single* quasi-linear schedule which minimizes the completion time of the whole computation (i.e., the time for $\vec{I}_l$).

Since the last computation point $\vec{I}_l$ usually lies close to the boundary (or even on the boundary), Theorem 7.2 is not directly applicable for computation at $\vec{I}_l$. In the following, however, we prove that $m'(\vec{I}_l)$ is still bounded to $f(\vec{I}_l)$ within a constant independent of $\vec{b}$.

Lemma 7.5

For any $\vec{I} \in \mathcal{P}_s$, $f(\vec{I}_l) \geq M'(\vec{I})$ where $M'(\vec{I})$ is defined in the proof of Thm. 7.2. □

Proof

Since $\vec{I}_l$ is the only last computation point in $\mathcal{P}$ and $\mathcal{P}$ is a finite polytope, for any $\vec{I} \in \mathcal{P}_s$, $\vec{I}_l \xrightarrow{L}_{\mathcal{P}} I$ for some integer $L > 0$. This implies $f(\vec{I}_l) \geq f(\vec{I}) + L$. Let $u_1, \ldots, u_k \geq 0$ be the optimum solution for $M'(\vec{I})$. Based on Lemma 7.4, $\vec{I} \xrightarrow{N}_{\mathcal{P}} \vec{I} - \sum_{i=1}^{k} u_i \vec{d}_i$ where $N = M'(\vec{I}) = \sum_{i=1}^{k} u_i$. It follows that $f(\vec{I}) \geq M'(\vec{I})$. Hence, $f(\vec{I}_l) \geq M'(\vec{I})$. □

Lemma 7.6

For any $\vec{I} \in \mathcal{P}_s$, there exists a constant K_1 such that $m(\vec{I}) - M'(\vec{I}) \leq K_1$.

□

Proof

We prove that $m(\vec{I}) - M(\vec{I}) \leq K'$ for some constant K' first.

Since $m(\vec{I}) = \max\left\{\vec{1}^t\vec{u} \middle| \begin{pmatrix} AD \\ -E_k \end{pmatrix} \vec{u} \leq \begin{pmatrix} A\vec{I} - \vec{b} \\ 0 \end{pmatrix}\right\}$ and

$M(\vec{I}) = \max\left\{\vec{1}u \middle| \begin{pmatrix} AD \\ -E_k \end{pmatrix} \vec{u} \leq \begin{pmatrix} A\vec{I} - \vec{b} - \vec{\pi} \\ 0 \end{pmatrix}\right\}$, based on a well known result on the sensitivity analysis in linear programming (see, Theorem 10.5, page 126 in [107]), for the optimum solution $\vec{u}'$ of $m(\vec{I})$ and the optimum solution $\vec{u}''$ of $M(\vec{I})$, $\|\vec{u}' - \vec{u}''\|_\infty \leq n\beta \| \begin{pmatrix} A\vec{I} - \vec{b} \\ 0 \end{pmatrix} - \begin{pmatrix} A\vec{I} - \vec{b} - \vec{\pi} \\ 0 \end{pmatrix} \|_\infty$ where β is the upper bound of all the entries in B^{-1} for each nonsingular submatrix B of $\begin{pmatrix} AD \\ -E_k \end{pmatrix}$. Hence, there exists a constant K' which is independent of $\vec{b}$ such that $m(\vec{I}) - M(\vec{I}) = \vec{1}^t(\vec{u}' - \vec{u}'') \leq K'$.

Since $M(\vec{I}) - M'(\vec{I}) \leq C_1$, $m(\vec{I}) - M'(\vec{I}) \leq m(\vec{I}) - M(\vec{I}) + C_1 \leq K' + C_1$.

Let $K_1 = K' + C_1$. We prove the lemma. □

Lemma 7.7

There exists an $\vec{I}_0 \in \mathcal{P}_s$ such that $m(\vec{I}_l) - m(\vec{I}_0) \leq K_2$ for some constant K_2 which is independent of $\vec{b}$. □

Proof

Consider two linear programming problems, $\max\{\vec{0}^t\vec{I} | (-A)\vec{I} \leq -\vec{b}\}$ and $\max\{\vec{0}^t\vec{I} | (-A)\vec{I} \leq -(\vec{b}+\vec{\pi})\}$. $\vec{I}_l$ is a feasible and optimum solution to the first problem. Based on similar argument in the proof of Lemma 7.6, there exists an optimum (feasible) solution $\vec{I}_0$ to the second problem such that $\|\vec{I}_l - \vec{I}_0\|_\infty \leq K' = n\beta\|\vec{\pi}\|_\infty$ where β is a constant independent of $\vec{b}$. For $\vec{I}_0$, there exists a $\vec{\lambda}' \in \Delta$ where Δ is the feasible region of (II') such that $m(\vec{I}_0) = \vec{\lambda}'^t(A\vec{I}_0 - \vec{b} - \vec{\pi})$. But $m(\vec{I}_l) \leq \vec{\lambda}'^t(A\vec{I}_l - \vec{b})$, we have $m(\vec{I}_l) - m(\vec{I}_0) \leq \vec{\lambda}'^t(A(\vec{I}_l - \vec{I}_0) + \vec{\pi})$. But $\|\vec{I}_l - \vec{I}_0\|_\infty \leq K'$, and so there is a constant K_2 such that $m(\vec{I}_l) - m(\vec{I}_0) \leq K_2$. □

Theorem 7.3

If $\vec{I}_l$ is the only last computation point in $\mathcal{P}$ for URE $\mathcal{U}$, then there exists a constant K independent of $\vec{b}$ such that $m'(\vec{I}_l) - f(\vec{I}_l) \leq K$. □

Proof

From Lemma 7.5 and Lemma 7.6, we have

$$\begin{aligned} m'(\vec{I}_l) - f(\vec{I}_l) &\leq m(\vec{I}_l) - \max_{\vec{I}\in\mathcal{P}_s}\{M'(\vec{I})\} = \min_{\vec{I}\in\mathcal{P}_s}\{m(\vec{I}_l) - M'(\vec{I})\} \\ &= \min_{\vec{I}\in\mathcal{P}_s}\{m(\vec{I}_l) - m(\vec{I}) + m(\vec{I}) - M'(\vec{I})\} \\ &\leq \min_{I\in\mathcal{P}_s}\{m(\vec{I}_l) - m(\vec{I})\} + K_1 \end{aligned}$$

Based on Lemma 7.7, there exists an $\vec{I}_0 \in \mathcal{P}_s$ such that $\min_{\vec{I}\in\mathcal{P}_s}\{m'(\vec{I}_l) -$

$m(\vec{I})\} \leq m(\vec{I}_l) - m(\vec{I}_0) \leq K_2$, letting $K = K_1 + K_2$, we prove the theorem.

□

Therefore, we can first find an optimum solution $\vec{\lambda}$ and α for $\vec{I}_l$ in the linear programming problem (III) which minimizes $m(\vec{I}_l)$. Based on Corollary 1, the quasi-linear function $L(\vec{I}) = \lfloor \vec{\lambda}^t I + \alpha \rfloor$ is also a schedule. Since $L(\vec{I}_l) = m'(\vec{I}_l)$, $L(\vec{I})$ minimizes the execution time of computation at $\vec{I}_l$ and thus minimizes the completion time of the whole computation.

Conclusions

In this chapter, we established results for scheduling a URE over a family of *extendible* integral polyhedra. We first showed that the relationship between the computability and the existence of a quasi-linear schedule. We then derived constant bounds based on the condition that the URE admits a quasi-linear schedule for any domain considered. The results on the constant bounds can be also derived based on the condition of computability.

The constants we derived to bound the schedule $m'(\vec{I})$ are in fact exponential in the dimension of the domain (i.e., n), although in practice this should not be a problem (n is usually rather small such as 3,4 or 5).

An important open problem is to extend our results for finding a single schedule which minimizes the whole computation for the case where there are more than one last computation points.

Compared with the result given by Darte, Khachiyan and Robert [32], our results characterize the nature of the free schedule function, while the result in [32] is only about the optimality of the total completion time. Furthermore, since only a

very limited domain, $\{\vec{I}|A\vec{I} \leq N\vec{b}\}$ where N is an integer as the domain parameter, was considered in [32], our results are applicable to a wider range of domains (in fact, *any* family of convex domains). For example, the following domain $\{(i,j)|0 \leq i \leq N1, 0 \leq j \leq 2N-1\}$ can not be represented as the domain considered in [32]. It should be noted that, however, the result presented in [32] characterizes the total completion time for domains $\{\vec{I}|A\vec{I} \leq N\vec{b}\}$ while our result on the total completion time is only applicable to a single last computation point.

CHAPTER VIII

LINEAR ALLOCATION FUNCTIONS FOR SYSTOLIC ARRAY DESIGN

Introduction

In practice, in the design of a systolic array, the user is often interested in arrays which are optimal with respect to a number of criteria such as the total computation time, the number of processors and the block pipeline rate [68], or even the amount of interstage data movement in a multistage systolic array. Some of these criteria, such as the total computation time, depend exclusively on the timing function. Some, such as the processor count, depend on the allocation function alone, while most others depend on a combination of the timing and allocation functions. The problem of finding an optimal (with respect to the computation time) timing function has been studied extensively [98] [109] and under some standard assumptions, can be formulated as a linear programming problem (or a sequence of linear programming problems).

For an n-dimensional recurrence, the number of valid linear allocation functions is infinite, even for a finite problem domain. This implies that it is impossible to enumerate all possible allocation functions. Hence, for optimizing performance criteria that depend on the allocation function, researches have been forced to develop strategies for pruning the search space of allocation functions. These strategies have been specific to the particular criterion. For example, in designing arrays with a minimal processor count, Wong and Delosme [118] develop and utilize an upper bound of the

length of the optimal projection vector (a linear allocation function can be uniquely represented by a projection vector). Thus the projection vectors can be generated in a sequence of nondecreasing length, and when the length exceeds the bound, the procedure can stop and claim that the best solution produced so far is the optimal one. In general, to develop such a strategy for a particular performance criterion, one must be able to systematically generate the allocation functions in an order by which one can guarantee that the optimal solution will be found. It is not always clear how to find such a strategy for any given performance criterion, nor is it usually easy to combine the strategies, if multiple criteria are being considered.

It is therefore very important to clearly understand the nature of the space of linear allocation functions, and to first prune it as much as possible *independently of the performance criterion.* In this chapter, we obtain upper bounds on the number of possible allocation functions, based on the following constraint: the interconnection links of the derived arrays must belong to a (usually small, and always finite) set of *permissible interconnections.* The bounds that we obtain are surprisingly low: there can be no more than 4 linear systolic implementations of 2-dimensional recurrences, and no more than 13 planar (purely systolic) arrays for a 3-dimensional system of recurrences. If diagonal connections are not permitted the number is 9, and if eight nearest neighbors are allowed, it is 25. For an arbitrary set of permissible interconnection vectors, we also develop an algorithm to determine the set of distinct "topologies" that can be constructed from these interconnections. We then show how these bounds can be used to systematically generate all allocation functions for a given system of UREs. This is achieved by introducing a normal form for these topologies. The average time complexity of the procedure is of the same order as the bound, which is the

best that we can expect to do. We conclude this introduction by formally describing the problem and then giving an outline of the chapter.

Moldovan, [85] used the *permissible interconnection matrix* in a very early paper, and a number of authors have proposed similar methods. Recently, Kothari et al. have also viewed the choice of allocation function as the solution of systems of diophantine equations [64]. Both these methods require manual inspection of the derived arrays to remove duplicates. We will see that these correspond to precisely the *unimodular affine transformations* of their allocation functions.

Notations and Problem Definition

For the purposes of this chapter, a system of UREs is completely described by a set of constant dependency vectors, $\{d_1, d_2, \ldots, d_k\}$ $(d_i \in \mathcal{Z}^n)$, and a convex polyhedral domain, $\mathcal{D}$. We denote by $\Delta = [d_1 \mid d_2 \mid \ldots \mid d_k]$ the $n \times k$ matrix formed from all the dependency vectors. The timing function is determined by an integral n-vector, λ, and the allocation function is an $(n-1) \times n$ integral matrix, A. There are two constraints that A must satisfy as stated in Chapter VI, namely, *Non-conflict* and **Dense array**.

Each dependency d_i of the URE is mapped to an interconnection link, $\gamma_i = Ad_i$ in the target array, and we define the *interconnection matrix*, as follows.

$$\Gamma = A\Delta = [\gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k] \qquad \text{(VIII.1)}$$

Although there are apparently $n^2 - n$ degrees of freedom in choosing A (the number of elements in the matrix A), this is not really true. Many different matrices yield arrays that are *equivalent* in that they are just a relabeling of the processors. It is well

known [99] that A is fully determined by a **projection vector**, u. Any two matrices, A and A' which satisfy $Au = A'u = 0$, yield arrays that are equivalent. Moreover, the non-conflict constraint can be satisfied simply by ensuring that $\lambda^T u \neq 0$. Other than this, any choice of u yields exactly one distinct array (some care must be taken to ensure that u is *reduced* (the gcd of all its elements is 1), and has a **positive leading element**). There are thus infinitely many valid allocation functions for a given system of UREs.

This approach for choosing the allocation functions does not take into account a very common, important constraint. The interconnections in the derived arrays are often required to belong to a set $\mathcal{P}$, of permissible interconnections. The following are four commonly used candidates for $\mathcal{P}$, representing respectively, the linear array, the two-dimensional mesh (four neighbors), the mesh with two diagonals (i.e., hexagonal arrays), and the mesh with eight neighbors.

$$\mathcal{P}_1 = \{0, \pm 1\} \tag{VIII.2}$$

$$\mathcal{P}_2 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \pm \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \pm \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\} \tag{VIII.3}$$

$$\mathcal{P}_3 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \pm \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \pm \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \pm \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\} \tag{VIII.4}$$

$$\mathcal{P}_4 = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \pm \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \pm \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \pm \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \pm \begin{bmatrix} 1 \\ -1 \end{bmatrix} \right\} \tag{VIII.5}$$

If we impose an additional constraint that for each dependency, d_i, $Ad_i = \gamma_i$ must belong to $\mathcal{P}$, we no longer have the freedom to choose *any* matrix that satisfies

$Au = 0$ as our allocation function. Indeed, for many values of u, there may be no A for which $\gamma_i = Ad_i \in \mathcal{P}$ for $i = 1 \ldots k$. For others, only *some* of the matrices satisfying $Au = 0$ may yield an array with permissible interconnections. We may now state our problem as follows. Given a set $\mathcal{P}$ of permissible interconnections, develop a systematic procedure to choose valid allocation functions (i.e., satisfying the non-conflict and dense array constraints) for a URE that will yield arrays whose interconnections belong to $\mathcal{P}$.

Our approach to tackle this problem, and the organization of the chapter, is as follows. First, in Sec VIII we investigate the properties of valid interconnections for a systolic array by using integral matrix theory. We also introduce the notion of congruence and similarity relations. In Sec VIII, we give a procedure to enumerate all possible "topologies" for a given permissible interconnection set, and in Sec VIII we show that the bounds for the above four common interconnection sets (i.e., P_1, P_2, P_3, P_4) are fairly small. Then, in Sec VIII we develop procedures to utilize these bounds to systematically generate all allocation functions for a given system of UREs. We show how we can use "normal forms" to reduce the time complexity of the procedure to the same order as the number of bounds we derive. Finally we discuss the implications of these results on the development of practical CAD tools for optimal systolic arrays.

Notation

We shall now introduce some formal notation which will be used throughout this chapter. Many of the ideas are fairly well known, but it is essential to treat them rigorously in order to explain the later development. First, we extend the standard

definition of unimodularity* to non-square matrices.

Definition 8.1

A $m \times n$ matrix U $(m \leq n)$ is said to be *e-unimodular* (for extended unimodular) if the gcd of the determinants of all its $m \times m$ submatrices is 1. □

It is well known [107] (pp. 47, Cor 4.1c), that a system of diophantine equations $Ux = I$ has an integral solution for *any* integral vector I, iff U is e-unimodular. Hence, the dense array constraint is satisfied iff the allocation function A is e-unimodular.

Lemma 8.1

The column Hermite form of an e-unimodular $m \times n$ matrix, A, is* $[E_m\ 0]$. □

Proof

By definition of column Hermite form there exists some integral unimodular C such that $AC = [L\ 0]$, where L is a non-negative lower triangular matrix whose diagonal entries are the unique maximal entries of the corresponding columns. Since column operations preserve the gcd of all order i subdeterminants, and A is e-unimodular, L must be unimodular. Hence all its diagonal entries are 1. Moreover, all other entries in any column are strictly less than this, i.e., 0. Hence, $L = E_m$. □

---

*A square matrix is unimodular if its determinant is ± 1.

*E_i denotes the $i \times i$ identity matrix.

Lemma 8.2

If A and B are two e-unimodular matrices, their product, AB is also e-unimodular. □

Proof

From Lemma 8.1 $B = [E_m\ 0]C^{-1}$ for some integral unimodular C. We have $AB = A[E_m\ 0]C^{-1} = [A\ 0]C^{-1}$. Hence, we need to show that $[A\ 0]$ is e-unimodular. This is obviously true, since adding any number of additional columns to an e-unimodular matrix still yields an e-unimodular matrix. □

Lemma 8.3

For two e-unimodular $m \times n$ matrices, A and B, if there exists a $m \times m$ rational matrix, U such that $A = UB$, then U must be integral and unimodular. □

Proof

From Lemma 8.1, $BC = [E_m\ 0]$ for some integral unimodular matrix. Hence, $AC = UBC = U[E_m\ 0] = [U\ 0]$. Since AC is integral, U must be integral. Furthermore, AC is e-unimodular, but the only $m \times m$ submatrix of $[U\ 0]$ whose determinant is *not* zero is U. Hence U must be unimodular.

□

We are interested in UREs whose computation graph (for a given parameter instance) is fully connected. If this were not so, the URE would describe a number of independent computations. Such a URE can always be rewritten as an equivalent one which has a connected computation graph. The following lemma gives necessary and sufficient conditions for this.

Lemma 8.4

The computation graph of a URE is connected iff the dependency matrix Δ is e-unimodular. □

Proof

The dag of the computation is connected if and only if that any index point I of the computation space $\mathcal{Z}^n$ can be represented by an integral linear combination of the dependency vectors $d_1, d_2, \ldots, d_k$ (i.e., every point is connected to the origin), i.e., the following equation has an integral solution L for any $p \in \mathcal{Z}^n$.

$$[d_1 d_2 \ldots d_k]L = p$$

This is true iff $\Delta = [d_1 \ d_2 \ \ldots \ d_k]$ is e-unimodular [107] (p.47, Corollary 4.1c). □

Analogously, if the set of permissible interconnections is not rich enough that we can express any integral point as a linear combination of the vectors in $\mathcal{P}$, then it is impossible to construct a dense array. Thus, we will henceforth assume that the matrix, P, whose columns are the elements of $\mathcal{P}$ is e-unimodular. The following relationships will be used to partition the interconnection matrices into classes which are essential to eliminate redundant candidates.

Definition 8.2

Two full row rank integral matrices, M_1 and M_2 are said to be *congruent* (denoted by $M_1 \doteq M_2$) if $UM_1 = M_2$ for some unimodular integral matrix, U. □

Definition 8.3

Two full row rank integral matrices, M_1 and M_2 are said to be *similar* (denoted by $M_1 \simeq M_2$) if $QM_1 = M_2$ for some non-singular rational matrix, Q. □

Note that both similarity and congruence are equivalence relations, and that congruence is a refinement of similarity.

Topological Equivalence

In the following, we give a mathematical property for two processor arrays to be topological equivalent. The property is fundamental for the derivation of bounds later on. First, let us recall the standard definitions of equivalence of processor arrays. A parallel architecture is described by a graph, the nodes denoting processor labels and edges denoting interconnections. Each processor has a finite number of *typed* I/O ports. All edges in the graph are also typed, so any interconnection in the array is between *similarly* typed ports on two processors. Two parallel architectures are defined to be *topologically equivalent* if their graphs are isomorphic to each other.

Regular processor arrays are parallel architectures that satisfy certain constraints. The processor labels are m-dimensional index vectors (moreover, *every* coordinate is a valid processor label), and the edges are of the form $p \leftrightarrow (p + \mu_i)$ for *all* processors, p, where the μ_i's are *constant*, m-dimensional vectors. We assume (for the present) that the space of the processors labels is infinite (there are no boundary processors), and all processors have the same number (say r) of I/O ports. Since all processors have identical interconnection links, associating a type to each edge of the graph is the same as typing the μ_i's. Hence the *topology* of the array is defined by an *ordered set*

of such constant vectors, or equivalently, an $m \times r$ integral matrix, $M = [\mu_1 \mid \ldots \mid \mu_r]$. We are interested in arrays that are connected, and so M must e-unimodular (using an argument similar to Lemma 8.4).

Theorem 8.1

Two regular processor arrays, $\mathcal{A}_1$ and $\mathcal{A}_2$ with topologies M_1 and M_2 respectively, are topologically equivalent (denoted by $\mathcal{A}_1 \equiv \mathcal{A}_2$), iff M_1 and M_2 are congruent to each other. □

Proof

Let $M_1 = [\mu_1, \ldots, \mu_r]$, and $M_2 = [\mu'_1, \ldots, \mu'_r]$.

If Part: Consider the linear transformation that maps any processor, p in $\mathcal{A}_1$ to $p' = Up$. Since every edge is of the form $p \leftrightarrow (p + \mu_i)$, it is mapped to $Up \leftrightarrow U(p+\mu_i)$ i.e., $Up \Leftrightarrow Up+U\mu_i$. The range of this transformation is the entire index space (since U is unimodular), and so this represents a regular processor array with the i-th ports of any processor, p, connected to $p+U\mu_i$. Since $U\mu_i$ is precisely the i-th column of M_2, this array is $\mathcal{A}_2$.

Only If Part: The two arrays are topologically equivalent, and hence there exists an isomorphism, say f between them. We first show that f must be linear. Any edge, $p \leftrightarrow (p+\mu_i)$ in $\mathcal{A}_1$ is mapped to $f(p) \leftrightarrow f(p+\mu_i)$ in $\mathcal{A}_2$, and since f is an isomorphism, this must be the edge $f(p) \leftrightarrow f(p) + \mu'_i$. Hence $f(p + \mu_i) = f(p) + \mu'_i$.

Similarly, $f(p + k\mu_i) = f(p) + k\mu'_i$ for any integer, k. Because M_1 is e-unimodular, any point p can be expressed as $\sum_{j=1}^{r} k_j \mu_j$, an integral linear

combination of its columns. Hence,

$$f(p) = f(\sum_{j=1}^{r} k_j \mu_j) = f(0 + \sum_{j=1}^{r} k_j \mu_j) = f(0) + \sum_{j=1}^{r} k_j \mu'_j$$

Hence, f must be linear and rational, and must be described by a rational $m \times m$ matrix. By Lemma 8.3, it must be integral and unimodular, i.e., $UM_1 = M_2$ for some integral unimodular $m \times m$ matrix U. □

Theorm 8.1 implies that any sequence of elementary row operations do not affect the topology of a regular processor array. However, elementary column operations are not permitted. Intuitively, this is so because the topology is defined as an *ordered* set of interconnections, and the order is crucial. For example, the two linear arrays $[1, 0]$ and $[0, 1]$ do not represent the same topology: a convolution array where the weights stay in the processors and the input values move is *not* the topologically the same as one where the weights move and the inputs stay.

Procedure 8.1

Given: Two $n \times k$ matrices, M_1 and M_2.

Output: true if $M_1 \doteq M_2$.

1. Determine an $n \times n$ non-singular submatrix, Γ_1 of M_1. If such a matrix does not exist, return *false.*
2. Check that the corresponding submatrix, Γ_2 of M_2 is also non-singular. If not, return *false.*
3. Determine $Q = \Gamma_2 \Gamma_1^{-1}$. If Q is not integral unimodular, return *false.*

4. If $QM_1 = M_2$ return *true*, else return *false*.

□

Procedure 1 above, is used to compare if two matrices are congruent. Note that similarity can be tested either by modifying the procedure (removing the test in step 3 above), or as follows. Compute the right null vector ν for the matrix, making sure that ν is chosen so that it is *reduced* and has a *positive leading element.* Then, similarity can be determined simply by comparing the respective ν's. Another method is to use a canonical form, and compare *these* for syntactic equality. This option will be discussed later. Note that if a regular processor array is derived from a URE $\{\Delta, \mathcal{D}\}$, by an allocation function A, its topology is $\Gamma = A\Delta$. Moreover, we have the following.

Remark 8.1

Two allocation functions A_1 and A_2 generate identical arrays iff the corresponding Γ_1 and Γ_2 are *topologically equivalent.* Indeed, $\Gamma_1 = U\Gamma_2$ iff $A_1 = UA_2$. □

Bounds on the Number of Allocation Functions

Our approach is based on the following simple observation. Instead of first choosing A and *deriving* Γ from it, if Γ is *given*, we can view Eqn (VIII.1) as a system of diophantine equations, and *solve* for A. This system has $(n-1) \times k$ equations (one for each element of Γ), and $n^2 - n$ unknowns in A (in fact, there are $n-1$ independent systems of equations, one for each row of A). Since n must be no greater than k for Δ to be e-unimodular, the system is fully (or over) determined, and yields a unique solution (if any) for A. Therefore, if we can enumerate the set of *all* such systems

of equations that can possibly occur this will constitute an bound on the number of allocation functions for the problem. Moreover, if this set is reasonably small, we will also have an effective synthesis procedure: each such system is solved to yield an array (if the system has no integral solution, we simply move on to the next one).

As a very crude approximation, we see that for a given problem, the set of all systems of equations of the form $A\Delta = \Gamma$ is precisely the set of all $(n-1) \times k$ matrices Γ that can be formed from the elements of $\mathcal{P}$. Hence, it is easy to see that the number of possible linear allocation functions is no more than the *number* of the system of equations. However, this is fairly large ($|\mathcal{P}|^k$), and we should use additional constraints that can reduce the size of this set. In particular, we know that these Γ's must represent valid interconnection matrices, and hence must be e-unimodular. Moreover, many such matrices represent topologically equivalent arrays. This yields the following procedure to determine the set of candidate equations of the form of (VIII.1).

Procedure 8.2

Given: A set $\mathcal{P}$ of permissible interconnections.

Output: A set $\mathcal{S}_k^e$ of all interconnection matrices that represent distinct arrays.

1. Construct the set, $\mathcal{S}_k$ of all e-unimodular $(n-1) \times k$ matrices whose columns belong to $\mathcal{P}$.
2. Partition $\mathcal{S}_k$ into equivalence classes under $\doteq$, and let $\mathcal{S}_k^e = \{\Gamma_i\}$ where Γ_i is a representative of each class. $\mathcal{S}_k^e$ is constructed incre-

mentally by comparing each candidate from $\mathcal{S}_k$ with the elements of (the partially constructed) $\mathcal{S}_k^c$, and adding it if it is distinct from the ones so far (as in the sieve of Eratosthenes).

□

Proc 2 runs in $O(|\mathcal{P}|^n|\mathcal{S}^c|)$. For *arbitrary* $\mathcal{P}$ this is as bad as $O(|\mathcal{P}|^{2n})$ (we can construct pathological cases where $|\mathcal{S}^c| = |\mathcal{S}|$). However, most commonly occurring sets of permissible interconnections have much more regularity. Thus the set of candidate interconnection matrices is the set of e-unimodular $(n-1) \times k$ matrices whose columns are in $\mathcal{P}$, under the equivalence partition induced by congruence. Hence an upper bound on the number of possible allocation functions is simply $|\mathcal{S}_k^c|$. This bound depends on k, the number of dependency vectors in the URE. In addition, when k grows, it can grow very fast. For example, when the permissible interconnection set is $\mathcal{P}_4$ defined in Section 158, $|\mathcal{S}_3^c|$ is 25 but $|\mathcal{S}_4^c|$ is 349! Therefore, if we intend to use these bounds to systematically generate all the arrays, we need to further reduce the bounds.

By observing that Eqn VIII.1 depends on Δ, the dependency matrix, we can further tighten this bound as follows. First, we notice that the dependency matrix Δ must be e-unimodular. Since e-unimodularity implies full row rank, Δ must contain an $n \times n$ non-singular submatrix, say Δ_1, (without loss of generality, we assume that Δ_1 consists of the first n columns of Δ). Let $\Delta = [\Delta_1 \mid \Delta_2]$, and correspondingly, $\Gamma = [\Gamma_1 \mid \Gamma_2]$. Equation (VIII.1) may therefore be written as:

$$\Gamma_1 = A\Delta_1 \qquad \text{(VIII.6)}$$

$$\Gamma_2 = A\Delta_2 \qquad \text{(VIII.7)}$$

Since A is required to be e-unimodular, Γ_1 must also be of full rank. Note that Γ_1 and Γ_2 are not necessarily e-unimodular themselves (similarly for Δ_1 and Δ_2). But (VIII.6) is fully determined, and can yield a solution for A by itself. We can now tighten our bound as follows.

Theorem 8.2

Let Γ and Γ' be two candidate interconnection matrices in $\mathcal{S}_k$ i.e., $\Gamma = [\Gamma_1 \mid \Gamma_2]$ and $\Gamma' = [\Gamma'_1 \mid \Gamma'_2]$ where Γ_1 and Γ'_1 are of full row rank. If $\Gamma_1 \simeq \Gamma'_1$ then Γ and Γ' cannot yield distinct allocation functions. □

Proof

Since $\Gamma_1 \simeq \Gamma'_1$, $\Gamma'_1 = Q\Gamma_1$ for some non-singular rational matrix, Q. Let, if possible there be two allocation functions, A and A', induced by Γ and Γ', respectively. Then, since Eqn VIII.6 is fully determined,

$$A = \Gamma_1\Delta_1^{-1}$$
$$A' = \Gamma'_1\Delta_1^{-1} = Q\Gamma_1\Delta_1^{-1} = QA$$

and A and A' must be e-unimodular (even though Δ_1^{-1} may not be integral). Hence, by Lemma 8.3, Q must be integral unimodular, i.e., the two allocation functions are not distinct. □

Hence, the number of distinct allocation functions is no more than the number of equivalence partitions of the set of all full row rank $(n-1) \times n$ matrices whose columns are in $\mathcal{P}$, under the *similarity* relation. This is denoted by $\mathcal{S}^s$. Two points

should be mentioned regarding this bound. First, for an *arbitrary* (finite) $\mathcal{P}$, it is not always possible that, given an $(n-1) \times n$ full row rank matrix whose columns belong to $\mathcal{P}$, we can always find $k-n$ additional column vectors belonging to $\mathcal{P}$ such that they form an *e-unimodular* $(n-1) \times k$ matrix. This implies that the bound may not be tight, i.e., for every element in $\mathcal{S}^s$, there may not be a candidate solution Γ. However, for many common cases such as $\mathcal{P}_1, \ldots, \mathcal{P}_4$, this is always possible. Second, since the bound is independent of the actual dependency matrix, it is also possible that for an $(n-1) \times n$ full row rank matrix Γ_1 (partition), there is no valid allocation functions for all e-unimodular $(n-1) \times k$ matrices whose submatrix of the first n columns is similar to Γ_1. In the following, however, we will show that for the degenerate case where $\mathcal{P}$ is Z^{n-1} (i.e., any integral $n-1$ dimensional vector is a permissible interconnection), one can *always* find an e-unimodular solution to Eqn VIII.1 for *any* $\Gamma_1 \in \mathcal{S}^s$. This indicates that our bound is tight for arbitrary $\mathcal{P}$. The following lemma first shows that any integral matrix is similar to some e-unimodular matrix (and that the similarity transformation involved is integral).

Lemma 8.5

Any $(n-1) \times n$ integral matrix Γ with full row rank is similar to some unimodular matrix Γ'. Moreover the matrix, T such that $\Gamma = T\Gamma'$, is integral. □

Proof

Any integral matrix Γ with full row rank can be transformed into its Smith Form by elementary row and column operations [107], i.e., $R\Gamma C = S = [D0] = D[I_{n-1}0]$, where R and C represent elementary row and column operations, and $D = \text{diag}(\delta_1, \delta_2, \ldots, \delta_{n-1})$ is a diagonal matrix. Therefore,

$\Gamma = R^{-1}D[I_{n-1}0]C^{-1}$. Take, $T = R^{-1}D$, and $\Gamma' = [I_{n-1}0]C^{-1}$. Since elementary column operations do not change the gcd of the determinants of the submatrices and because $[I_{n-1}0]$ is unimodular, Γ' is unimodular too. □

As an aside, the above Lemma indicates that if we restrict our similarity relation to only integral transformations, then, this relation is a strict partial order. Any set of (full row rank) matrices that are congruent to each other has a subset which are *minimal* under this partial order. Moreover, if the set is *closed* under congruence, the minimal subset is a singleton. The following theorem gives us a method to determine an e-unimodular solution to $A\Delta_1 = \Gamma_1'$, for some Γ_1' similar to any candidate Γ_1.

Theorem 8.3

For any candidate interconnection matrix $\Gamma_1 \in \mathcal{S}^s$, there exists $\Gamma_1' \simeq \Gamma_1$, such that $A\Delta_1 = \Gamma_1'$ has an e-unimodular solution. □

Proof

The following procedure yields the desired solution.

- Solve VIII.6 as a system of linear (rather than diophantine) equations. So, $A = \Gamma_1\Delta_1^{-1}$ which is, in general, rational.
- Let A' be the integral matrix obtained by multiplying A by d, the least common multiple of the denominators of all the entries of A
- By Lemma 8.5 there exists e-unimodular A'' such that $A' = TA''$, and T is a non-singular integral matrix.

Hence $\Gamma_1' = dT^{-1}\Gamma_1$ is similar to Γ_1 and A'' is the desired e-unimodular

solution (we say $V = dT^{-1}$). Moreover, since our proof is constructive, we can determine T and A''. □

So far, all we have are the upper bounds for the number of allocation functions. The problem of effectively utilizing these bounds to systematically generate all the valid allocation functions still remains to be solved. This is not at all obvious, and in Sec VIII, we will develop a procedure to generate all the valid allocation functions which has space and time complexity of $O(|\mathcal{S}_k^c|)$. This can be improved by using various indexing schemes, and standard data structures for searching. Before we discuss this, we first show that for the common cases defined in Sec VIII, the bounds are surprisingly small.

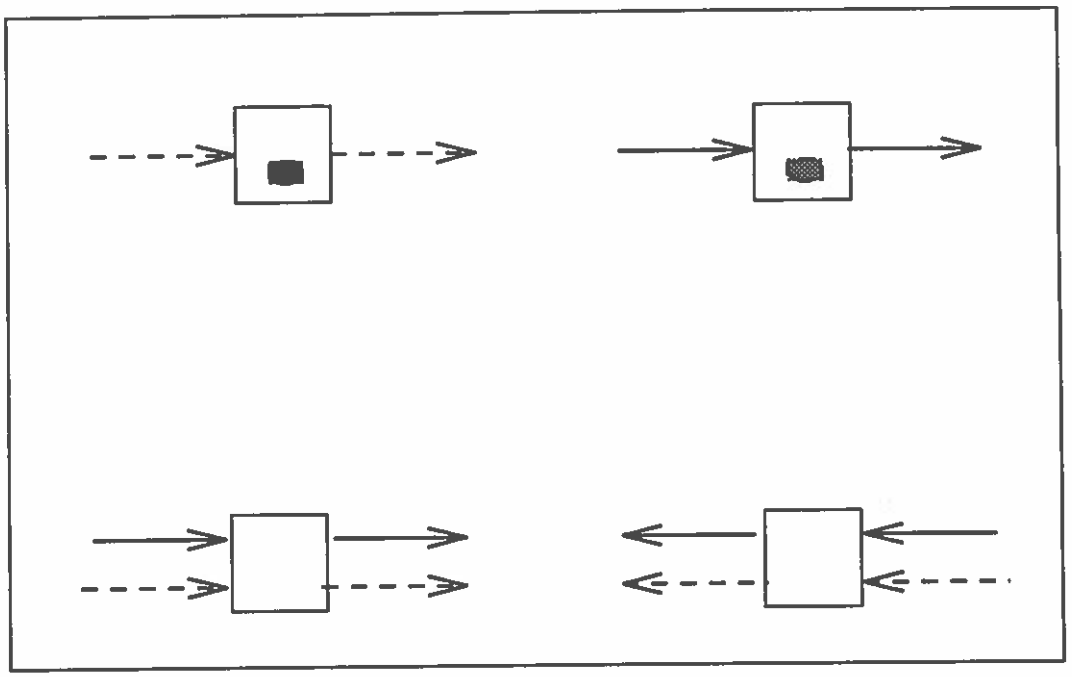
Interconnection Matrices for the Common Cases

Applying the above procedure to the standard sets of permissible interconnections (Equations VIII.2–VIII.4) we obtain the following results. There can be no more than 4 linear systolic arrays with nearest neighbor interconnections for any two-dimensional URE. These correspond to the topologies given as follows (see Figure 42).

$$\mathcal{S}_1^s = \{[01], [10], [11], [1-1]\}$$

For $\mathcal{P}_2$, we have the following nine possible topologies (see Figure 43).

$$\mathcal{S}_2^s = \left\{ \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \right.$$
$$\left. \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right\}$$



One data value (solid) stays in the processor while the other one (gray) moves, the gray one stays while the solid one moves, both of them move in the same direction, or they both move in opposite directions.

Figure 42: The Only Four Linear Arrays That Can Be Derived from a Two-Dimensional Recurrence.

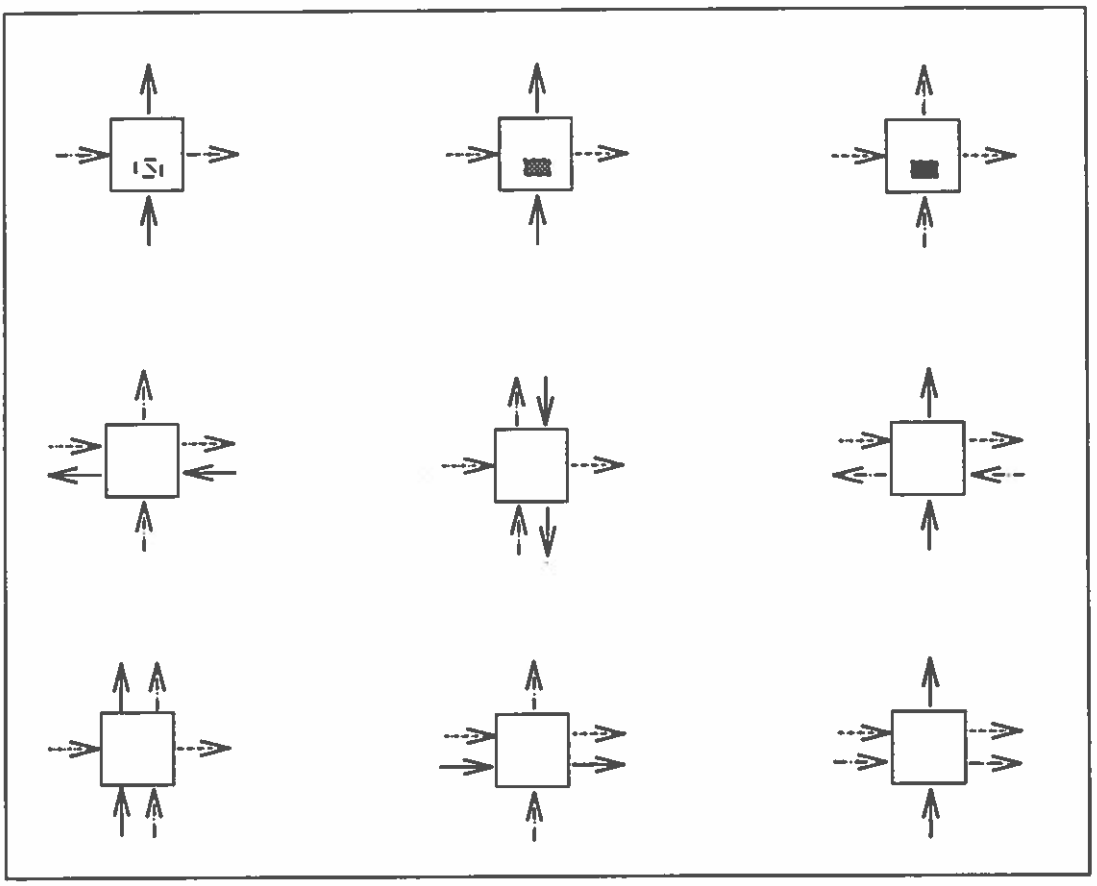


Figure 43: All Distinct Two-Dimensional Arrays with Pure Mesh Connections.

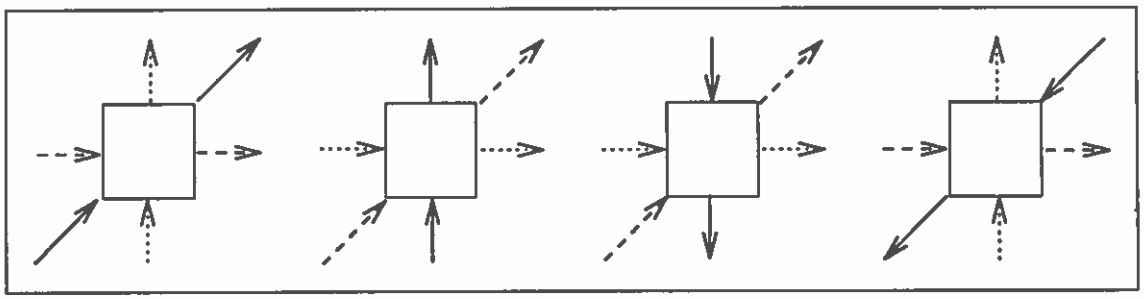


Figure 44: Additional Two-Dimensional Arrays If One Set of Diagonals Are Permitted ($\mathcal{P}_3$).

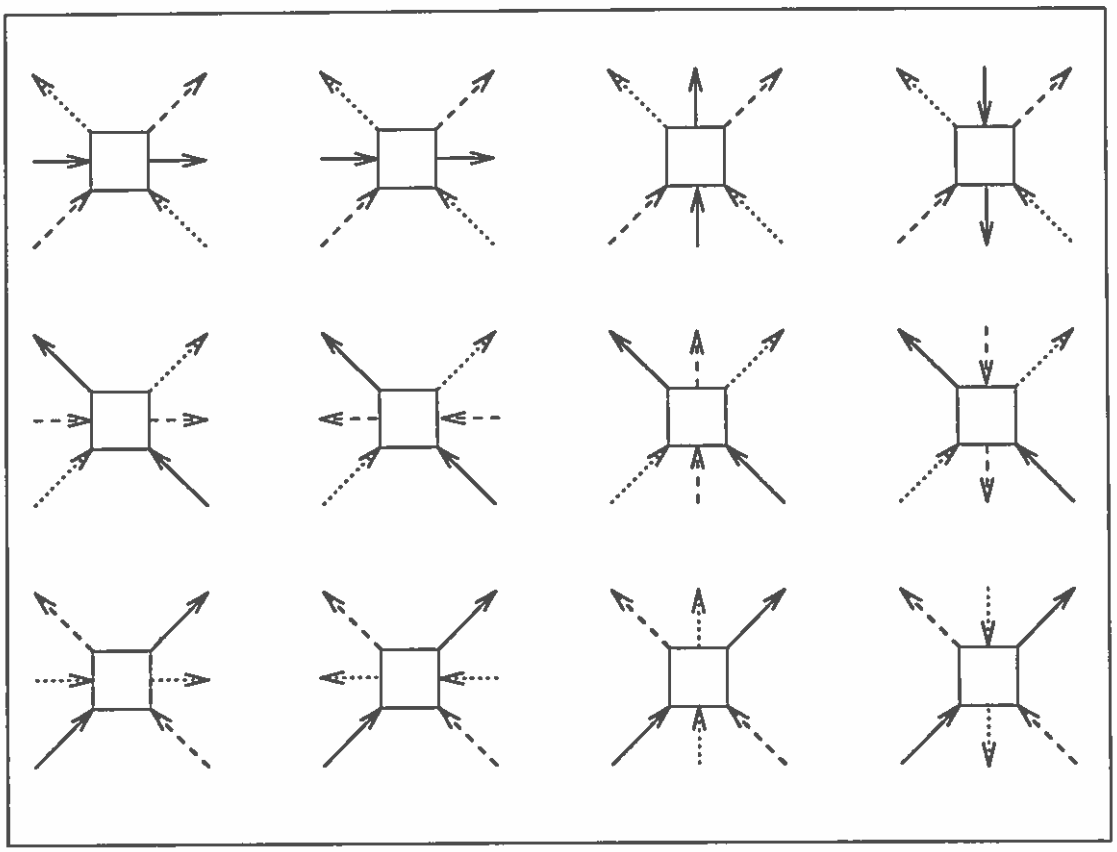


Figure 45: Additional Two-Dimensional Arrays for Eight Nearest Neighbors, ($\mathcal{P}_4$).

If hexagonal connections are permitted (i.e., for $\mathcal{P}_3$) the following 4 topologies are possible in addition to $\mathcal{S}_2^s$, as shown in Figure 44.

$$\mathcal{S}_3^s = \mathcal{S}_2^s \cup \left\{ \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \right\}$$

For eight nearest neighbors, i.e. $\mathcal{P}_4$ there are twelve additional interconnections as shown in Figure 45.

$$\mathcal{S}_4^s = \mathcal{S}_3^s \cup \left\{ \begin{bmatrix} 1 & -1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & -1 & -1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & -1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 0 \\ 1 & 1 & -1 \end{bmatrix} \right.$$
$$\begin{bmatrix} 1 & 1 & -1 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} -1 & 1 & -1 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 & -1 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 & -1 \\ -1 & 1 & -1 \end{bmatrix}$$
$$\left. \begin{bmatrix} -1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & -1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \\ 1 & -1 & 1 \end{bmatrix} \right\}$$

Note that while $\mathcal{S}^s(\mathcal{P}_4)$ has only 25 elements as shown above, $\mathcal{S}_4^c(\mathcal{P}_4)$ has 349 as can be verified by running Proc 2. This grows very large as k increases and so it is very desirable to obtain a procedure that fully utilizes the tighter bounds.

Systematic Derivation of Valid Allocation Functions

In this section, we describe systematic procedures to utilize the bounds obtained in Sec VIII to generate all the valid allocation functions for a problem. First of all, there is a naive procedure to utilize the bounds given by the congruence relation (i.e., $\mathcal{S}_k^c$). The procedure simply tries to solve Eqn VIII.1 for each $\Gamma \in \mathcal{S}_k^c$. Whenever an e-unimodular solution A is found, it represents a valid allocation function. This

procedure runs in time proportional to $|\mathcal{S}_k^e|$, and has space complexity of the same order of magnitude. Also note that since we have assumed that the Δ_1 is of full row rank (which is always possible since we can rearrange the dependency vectors), Γ_1 must be of full row rank. Hence, we can immediately discard those elements of $\mathcal{S}_k^e$, whose first n columns are not of full row rank. We call this set the reduced $\mathcal{S}_k^e$.

Improving this procedure to utilize our tighter bound is not as straightforward as it may seem. As we saw on page 167, we can first partition the reduced set $\mathcal{S}_k^e$ by the similarity relation according to the submatrices formed by the first n columns. Formally, we define the partition* as follows.

Definition 8.4

> For any two elements Γ and Γ' in the reduced set $\mathcal{S}_k^e$, they are in the same partition class iff $\Gamma_1 \simeq \Gamma_1'$ where Γ and Γ_1' are the submatrices of the first n columns of Γ and Γ' respectively. We call this partition $\mathcal{C}$. □

Clearly, $|\mathcal{C}| \leq |\mathcal{S}^s|$. Also note that two matrices Γ and Γ' being in the same partition class of $\mathcal{C}$ does not not necessarily imply that they are similar to each other, let alone congruent (only their first n columns are similar). This raises a problem when we try to improve the above procedure. Suppose that $\Gamma = [\Gamma_1\ \Gamma_2]$ is the representative of a class $\mathcal{R}$ of $\mathcal{C}$, and there is no integral e-unimodular solution A for $\Gamma_1 = A\Delta_1$ (however, because Γ_1 is full row rank, there will always exist a *rational* solution). We cannot simply discard Γ as a candidate that can generate a valid allocation function. It is possible, that a *rational* transformation of A may yield a valid allocation function for some other member Γ' of $\mathcal{R}$.

---

*A partition of a given set, S, denotes a set of disjoint subsets of S whose union is equal to S.

Example 8.1

Consider the two candidate interconnection matrices (whose columns are in $\mathcal{P}_4$)

$$\Gamma_1 = \begin{bmatrix} 1 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix} \quad \Gamma_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \text{and} \quad \Gamma_1' = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \Gamma_2' = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

and $\Gamma = [\Gamma_1\ \Gamma_2]$, $\Gamma' = [\Gamma_1'\ \Gamma_2']$. It is easy to see that $\Gamma_1 \simeq \Gamma_1'$, since

$$\Gamma_1 = T\Gamma_1'$$

where

$$T = \begin{bmatrix} 1 & 0 \\ -1 & 2 \end{bmatrix}$$

Hence, only one of them, say Γ' will be picked as the representative of the class which contains both Γ and Γ'. Now, consider a problem instance given by

$$\Delta_1 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad \Delta_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

and $\Delta = \Delta_1\Delta_2$. It is easy to verify that Δ is e-unimodular. Solving Equation $A\Delta_1 = \Delta_1'$ for A yields

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2} & 0 \end{bmatrix}$$

which is not even integral, let alone e-unimodular. But if we use Γ as the

representative of this equivalence class, the solution is

$$A = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{bmatrix}$$

It is easy to verify that A is indeed a valid allocation function for Γ. □

It is easy to see that, if two $(n-1) \times n$ matrices $\Gamma_1 \simeq \Gamma_1'$, the corresponding *rational* allocation matrices A and A' derived by solving Eqn VIII.6 are similar. Lemma 8.6 shows that we do not need to solve Eqn VIII.6 for A for each $\Gamma \in \mathcal{R}$.

Let $\mathcal{R}$ be an arbitrary partition class of $\mathcal{C}$, Γ be the representative of $\mathcal{R}$ and Γ_1 be the first n columns of Γ. Furthermore, we know by Theorem 8.3 there exists $\Gamma_1' \simeq \Gamma_1$ for which $A\Delta_1 = \Gamma_1'$ has an e-unimodular solution. Let $A_0 = V\Gamma_1\Delta_1^{-1}$ be this solution (see the proof of the theorem for details). We have the following lemma.

Lemma 8.6

There is a valid allocation function for class $\mathcal{R}$ iff $A_0\Delta$ is congruent to some element in $\mathcal{R}$. □

Proof

First of all, based on the proof of Theorem 8.3, A_0 is e-unimodular regardless of what Γ is. To prove the *if part*, suppose $A_0\Delta$ is similar to some element Γ' in $\mathcal{R}$, i.e., $UA_0\Delta = \Gamma'$ for some unimodular matrix U. It is easy to see that UA_0 is a valid allocation function for Γ' since UA_0 is unimodular.

Conversely, suppose that there exists a $\Gamma' = [\Gamma_1', \Gamma_2']$ in $\mathcal{R}$ such that there exists a valid allocation function A' (e-unimodular) for it. We prove that

$A_0\Delta$ is congruent to Γ'. Since $\Gamma_1 \simeq \Gamma'_1$, there exists a non-singular rational matrix T such that $A_0 = TA'$. But since A_0 and A' are e-unimodular, based on Lemma 8.3, T must be integral and unimodular. Thus, we know that $A_0\Delta$ is congruent to $A'\Delta = \Gamma'$. □

The above discussion implies that, in systematically generating allocation functions, we can not simply discard the elements in a partition class $\mathcal{R}$ represented by a single element. Therefore, it is difficult (if not impossible) to reduce the space complexity (remember that as the number of dependencies increases, $|\mathcal{S}_k^c|$ grows very fast). This difficulty rises from the fact that for any *finite* permissible interconnection set $\mathcal{P}$ is not closed under unimodular transformation. There is much room, however, to improve the time complexity of the procedure. The following is the general skeleton of the procedure to generate all valid allocation functions.

Procedure 8.3

Given: A set $\mathcal{P}$ of permissible interconnections, a dependency matrix Δ and a partition $\mathcal{C}$.

Output: All the valid allocation functions.

For each class $\mathcal{R} \in \mathcal{C}$,

1. Let Γ be the representative of $\mathcal{R}$.
2. Solve for A_0 as in Lemma 8.6.
3. If $U := \text{Test}(A_0, \mathcal{R})$ is not *false*, then UA_0 is the valid allocation function for $\mathcal{R}$, otherwise $\mathcal{R}$ does not yield a valid allocation function.

□

Here, Test checks whether $A_0\Delta$ is congruent to some elements in class $\mathcal{R}$ and returns *false* if it fails; otherwise, i.e., if it finds an element Γ such that $UA_0\Delta = \Gamma$ for some unimodular matrix U, it returns U.

There are two places to in Procedure 3 which need to be refined: one is how to choose a representative of the $\mathcal{R}$'s, and the other is how to implement function Test, which will dominate the time complexity of the algorithm (recall that for many common cases, $|\mathcal{C}|$, the number of the iterations in Procedure 3, is much smaller than than $|\mathcal{S}_k^c|$). In the following, we give several approaches.

The simplest approach is to randomly pick an element from $\mathcal{R}$ as the representative and implement function Test as follows: sequentially compare all the elements of $\mathcal{R}$ with $A_0\Delta = V\Gamma_1[I_{n-1}, \Delta_1^{-1}\Delta_2]$ using Proc 1. As soon as we find one successful match, we can skip the remaining elements of $\mathcal{R}$ and Test returns the corresponding element. This is safe, since in any partition class $\mathcal{R}$, at most one element can yield a valid allocation function. The worst case running time of Test is proportional to $|\mathcal{R}|$, and the average time is half of this.

The implementation of Test in the first approach can be further improved as follows. Note that any element in $\mathcal{R}$ can be written as $[Q\Gamma_1, \Gamma_2]$, for some non-singular rational matrix Q, which can be precomputed (as can Q^{-1}). To test whether $[Q\Gamma_1, \Gamma_2] \in \mathcal{R}$ is congruent to $A_0\Delta$, we have to see whether there is a unimodular transformation U such that $A_0\Delta = U[Q\Gamma_0, \Gamma_2]$. It is easy to show that $U = VQ^{-1}$, and so all we have to do is to test whether VQ^{-1} is unimodular, and whether $A_0\Delta_2 = VQ^{-1}\Gamma_2$. A necessary condition for U to be unimodular is that $\det(V) = \pm\det(Q)$. Hence we need not compare $A_0\Delta$ with *all* the elements of $\mathcal{R}$, only with those whose Q

component has the same (absolute) determinant as V. Thus, if we further partition $\mathcal{R}$ into blocks with $\det(Q)$ as a key, we can reduce the time complexity of Test. However, a linear search is still required *within* each block.

It is possible to improve this to logarithmic (or constant) time, based on the following observations. For each element of $\mathcal{R}$, our procedure Test performs an operation which tests for congruence. This is a binary comparison which simply returns true or false. If we were able to refine this to an *order relation*—if while determining whether two matrices were congruent, we were also able to determine which one was "greater"—we could then sort our candidates according to this order and use a binary search (or some hashing technique) to test for a match. We now introduce the concept of normal forms of interconnection matrices which enables us to devise such an order, and to dramatically reduce the average time complexity of the search procedure.

Normal Forms for Topologies

As described earlier, the particular candidate we use as our representative of each equivalence class under $\doteq$ is not unique, and may depend on the way our algorithm was implemented. It is useful to have a standard form which is a unique representative of each partition. Furthermore, since the sets of candidate interconnections $\mathcal{R}$ are precomputed, we can reduce the test for congruence to a simple syntactic test for equality if they are stored in such a form. We now derive such a canonical *normal form*, and show that it is unique. Since $\Gamma \doteq \Gamma'$ iff one of them can be reduced to the other using only elementary row operations, we shall obtain a normal form that is similar to Hermite normal form [107] except that rather than column operations, only row operations are permitted. We call this the *row-Hermite* normal form.

Theorem 8.4

Every $(n-1)\times n$ $(m < k)$ integral full row rank matrix Γ, is topologically equivalent to its row-Hermite normal form, $\Gamma^* = \begin{bmatrix} A_i & a_{i+1} & B \\ 0 & 0 & C \end{bmatrix}$, where $\begin{bmatrix} A_i & B \\ 0 & C \end{bmatrix} = S$ is a non-singular, non-negative upper-triangular matrix whose diagonal elements are the maximal entries in the corresponding column, and $\begin{bmatrix} a_{i+1} \\ 0 \end{bmatrix}$ is the $(i+1)$-th column of Γ^* (for some i) □

Proof

Our proof is based on the following induction. Suppose we have transformed Γ into the form $\begin{bmatrix} A_1 & A_2 \\ 0 & B_2 \end{bmatrix}$ where A_1 is upper triangular and with positive diagonal. We perform the following elementary row operations to B_2. Make the first column of B_2 non-negative (if the whole column is zero, proceed to the second column). This can always be done because we can change the signs of a row by multiplying by -1. Pick the smallest positive element s on this column (say it is in row l) and repeatedly subtract row l from the other rows until all their entries in this column are less than s. Repeat this, until all except one element in this column are zeroed out (this is similar to Euclid's algorithm). Exchange this row with the first row.

Since Γ is of full row rank, the case when all the entries in the first column of B_2 are zero will occur at most once. Hence, there are at most

two columns with the same "height" of nonzero elements, and all those elements on the diagonal are positive. The final matrix will have the "shape" that we desire, but its off-diagonal elements are not necessarily positive. This too can be easily accomplished by appropriate elementary row operations. The final form is Γ_*. □

Theorem 8.5

For any two full row rank $(n-1) \times n$ matrices Γ_1 and Γ_2, $\Gamma_1 \doteq \Gamma_2$ iff $\Gamma_1^* = \Gamma_2^*$. □

Proof

If Part: $\Gamma_1 \doteq \Gamma_1^* = \Gamma_2^* \doteq \Gamma_2 \Rightarrow \Gamma_1 \doteq \Gamma_2$ (by transitivity).

Only If Part: Let $\Gamma_1 \doteq \Gamma_2$, so $\Gamma_1 = U'\Gamma_2$ for some unimodular U'. Moreover, by definition of row-Hermite normal form, $\Gamma_1 \doteq \Gamma_1^*$ (so $\Gamma_1 = U_1\Gamma_1^*$) and $\Gamma_2 \doteq \Gamma_2^*$ (so $\Gamma_2 = U_2\Gamma_2^*$), for some U_1 and U_2. Hence $\Gamma_1^* = U_1^{-1}U'U_2\Gamma_2^* = U\Gamma_2^*$ for $U = U_1^{-1}U'U_2$. We now show that this implies the U is the identity matrix, i.e., $\Gamma_1^* = \Gamma_2^*$. Let

$$\Gamma_1^* = \begin{bmatrix} A_i & a_{i+1} & B \\ 0 & 0 & C \end{bmatrix}, \Gamma_2^* = \begin{bmatrix} A'_j & a'_{j+1} & B' \\ 0 & 0 & C' \end{bmatrix}, S = \begin{bmatrix} A_i & B \\ 0 & C \end{bmatrix} \text{ and}$$

$$S' = \begin{bmatrix} A'_j & B' \\ 0 & C' \end{bmatrix}$$

Without loss of generality, let $j \leq i$. We first prove that $i = j$. If it were not so, US is nonsingular, but the corresponding submatrix in Γ_2 includes $\begin{bmatrix} A'_j & a'_{j+1} \\ 0 & 0 \end{bmatrix}$ which makes S singular. Therefore, $i = j$. Now, consider $S' = US$. First of all, it is easy to see that U should be also upper triangular. Notice $u_{11}u_{22}\ldots u_{n-1,n-1} = 1$ (because U is unimodular), $u_{ll}s_{ll} = s'_{ll}$

(for $1 \leq l \leq n-1$) and $s_{ll}, s'_{ll} > 0$, we have $u_{ll} = 1$. Further, consider $s'_{l-1l} = s_{l-1l} + u_{l-1l}s_{ll}$. From $s_{l-1l} + u_{l-1l}s_{ll} \geq 0$, we have $u_{l-1l} \geq 0$ (because s_{ll} is larger than s_{l-1l}) and from $s'_{l-1l} < s'_{ll} = s_{ll}$, we have $u_{l-1l} = 0$. Now, consider $s'_{l-1l+1} = s_{l-1l+1} + u_{l-1l+1}s_{l+1l+1}$, using the same argument, we have $u_{l-1l+1} = 0$. Inductively, for $1 < k < l$, $u_{l-k,l} = 0$. Therefore, U is the identity matrix. □

The definition of normal forms can be trivially extended to $m \times k$ integral matrices where $m < k$, and the submatrix formed by the first $m+1$ columns is full row rank (all our candidate interconnection matrices in the reduced $\mathcal{S}_k^e$ are of this form). We simply convert the first $m+1$ columns to row-Hermite normal form, and apply the same transformations to the remaining columns. It is also easy to show that these normal forms are unique, and Theorem 8.5 can also be trivially extended.

We now improve the testing function Test in Proc. 3 as follows. We index elements in a partition class $\mathcal{R}$ of $\mathcal{C}$ by their row-Hermite normal form. To test whether $A_0\Delta$ is congruent to an element Γ in $\mathcal{R}$, we first convert $A_0\Delta$ into its normal form N and based on Theorem 8.5, the testing of whether $A_0\Delta$ congruent to Γ is equivalent to testing whether N is syntactically equal to the normal form of Γ (which is the key of Γ). Thus, the testing problem reduces to a conventional search of an ordered list, and Test can be implemented using standard techniques. A binary search tree improves the time to logarithmic, and hashing may also be used to obtain a constant time algorithm for Test. Proc. 3 for determining all the valid allocation functions will now run in $O(|\mathcal{S}^s|)$, which is a good as we can expect to do. Note that since the reduced $\mathcal{S}_k^e$ is not closed under congruence, the normal form may not itself be a permissible interconnection. In this case we also have to explicitly store the original

interconnection matrix in reduced $\mathcal{S}_k^c$ to return a correct unimodular transformation matrix U.

The Design of Optimal Systolic Arrays

The results presented in this chapter have direct applications in the design of optimal systolic arrays. We illustrate two such examples.

One of the most important criteria in systolic array design is the number of processors of the array. In practice, we intend to minimize the processor count because processors occupy precious resources such as silicon area on chip for the array. Under the conventional framework for systolic array synthesis, choosing different projection vectors (hence different linear allocation functions) yields different processor counts. In general, the processor count of the resultant array is the number of integral points of the image of the computation domain under the projection. It depends exclusively on the projection vector (linear allocation function) chosen for a given problem. Thus, to minimize the processor count, it is desirable to search for the linear allocation function which yields the minimal processor count. The number of valid linear allocation function for a given problem, however, is infinite even for a finite computation domain. Furthermore, since the processor count cannot be formulated as a linear function (except for two dimensional recurrences), linear programming techniques cannot be directly used as in the minimization of the total computation time (i.e. choosing the optimal timing function) [109].

Wong and Delosme [118] considered this problem and proposed a method to prune the search space of projection vectors. They proved that there exists an upper bound for the *length* of the projection vector, u (recall that u is the basis for the right-null space of the allocation matrix, A, i.e., a vector, such that $Au = 0$) which

yields the minimal processor count. This bound depends on the "shape" of the domain of computation, and Wong and Delosme give a constructive method to find the bound. Using this, one obtains a processor-minimal systolic array by enumerating all candidate projection vectors u that are smaller than (or equal to) this upper bound, and picking the one that yields the best array among only these. Note that since the bound depends on the shape of the computation domain, it may be different for different parameter sizes of the same problem.

In contrast with this, using our approach, one would construct all the arrays that can be derived for the given problem and using a given set of permissible interconnections (such as "pure systolic") using (Proc. 3). Then one would choose the optimal one by comparing the processor counts of each of the arrays. Because of the small bounds for the number of these valid allocation functions, we expect that such a procedure would be more efficient. Furthermore, the search space of valid linear allocation functions is independent of the problem size. One drawback of the method is that it is dependent on the particular choice of permissible interconnections (this is our premise). If a user is interested in the processor-minimal systolic array for a given problem, regardless of what the interconnections are (i.e., if $\mathcal{P} = \mathcal{Z}^{n-1}$) our procedure would be inapplicable.

In practice however, the absolute processor count may not be important. There are many different performance criteria that may be used in systolic design, including throughput, processor utilization, block pipelining rate, etc. [68], and many of these are closely related. One such measure is the processor pipelining rate α (if every processor is active in one out of every α clock cycles, the processor pipelining rate is α). It is well known that $\alpha = \lambda^T u$ (recall that λ^T is the schedule vector). Ideally, one

would like an array where $\alpha = 1$, it is also well known that by clustering adjacent processors together, one can often achieve this. This problem is addressed in Chapter IX where it is shown that such clustering can be deduced automatically. Clustering has the added advantage that the processor count is also reduced by a factor of α. Thus it would seem that the real cost measure that one should minimize while designing systolic arrays is not just V, the volume of the projection of the domain of computation, but V/α. It is not obvious how the method proposed by Wong and Delosme can be adapted to this new definition of processor count. Since our results enable the designer to systematically enumerate the (finite) space of all possible arrays that can be derived, it can be easily adapted to the new definition, and to any cost criterion (or indeed any combination of criteria) that the designer chooses.

Another application of the results in this chapter is in the design of optimal multistage systolic arrays. Many practical algorithms in signal processing and numerical analysis naturally have several different phases (i.e., there are several different nested loops in the algorithm). For example, a multiplication of three matrices can be decomposed into two multiplications of two matrices. There are two approaches to the array implementation of such algorithms. The first is to design different arrays for different stages and the other is to design a single systolic array for all the different stages. In the first approach, it is desirable to minimize the interstage data movement which are caused because of the mismatch of the input and output boundaries of the arrays for different stages. In the second approach, besides the minimization of the interstage data movement, it is also desirable to minimize the difference of interconnection structures caused by the difference of the computation domains for different stages. Such minimization problems require an exhaustive search in the spaces of

linear allocation functions for different stages. By using our results, an efficient procedure can be designed since the search space for each stage is quite small.

To summarize, the results in this chapter can be used to design optimal systolic arrays for *various* criteria even for some fairly complicated design problem. This is because our work shows that there exists an efficient procedure to systematically generate all possible *linear allocation functions*, which are essential to optimize many design criteria. Since the procedure is independent of different optimization criteria, such a method can be widely used.

Conclusions

We have shown that the problem of determining valid linear allocation functions for a system of UREs has only finitely many solutions, if one considers the fact that the desired arrays must have interconnections that belong to a (finite) set of permissible interconnection vectors (which is a very realistic assumption). Moreover, we have given an effective method for constructing a sufficiently tight upper bound on the *number* of distinct valid solutions that can ever be found. These bounds, for the common cases, are surprisingly small. By using the idea of normal forms, we also give a systematic procedure to enumerate all possible distinct valid allocation functions in a time complexity which is clearly the best we can do.

It should be noted that, although we have reduced the time complexity of the procedure to enumerate all distinct allocation functions to the utmost, based on the tight bound, S^s, we have to store S_k^c instead of S^s number of interconnection matrices. Therefore, our precomputed interconnection matrices are related to the number of dependencies k of the problem specification. One way to tackle this problem is to precompute S_n^c which depends only on n, the dimension of the problem domain and

then generate S_k^c on the fly. This will dramatically reduce the space required and make our procedure problem independent, at the price of additional time costs. Note that the determination of S_k^c is a one-time computation, so it may be done off line for the common cases and the on-the-fly approach can be used only when the system does not have this information (and this can be stored for later use).

Notice that in the process of enumerating all the allocation functions, the equations to be solved (i.e. the instances of Eqn VIII.1) all involve only Δ_1, the first n columns of the dependency matrix, Δ. We expect that some properties of Δ (for example, the determinant of Δ_1) could impose additional constraints on the space of candidate Γ's. By investigating these constraints, we may be able to further reduce time and space complexities of the enumerating procedure. This is currently under investigation.

Our results also raise another question in systolic array research, namely what are permissible interconnection structures for systolic arrays? One of the possible criteria is distance. However, even for this simple criterion, it is still worthwhile to investigate different notions of the distance (say, physical distance or the minimum number of integral points). We expect that under different criteria, there will be different upper bounds on the number of distinct valid allocation functions (and of course, different enumerating procedures).

CHAPTER IX

QUASI-LINEAR ALLOCATION FUNCTIONS FOR EFFICIENT ARRAY DESIGN

Introduction

In general, processors in a systolic array may not be active all the time during the lifetime of computation. In particular, a processor exhibits a periodical behavior: active at one out of every δ clock cycles. This introduces the concept of processor utilization or processor efficiency. In this chapter, we study the problem of deriving systolic arrays whose processors are always active, i.e. $\delta = 1$.

Traditionally, allocation functions have been restricted to **integral** linear projections (i.e., the allocation function was given by an integral $(n-1) \times n$ matrix). In this chapter, we study a class of functions called quasi-linear allocation functions. We describe the special properties of such functions and give conditions for them to be valid allocation functions. We also show how they can be used to derive arrays that have 100% efficiency.

It has been observed that by *merging* every δ neighboring processors which are active at different times, one can improve the *efficiency* of the array to 100% without slowing down the array and increasing the cost (each PE may have a few additional registers and some extra control, but *no* additional functional units are required). However, this technique is not systematic and it can be only used case by case. In this chapter, we prove that it is *always* possible to perform clustering in

such a manner that the efficiency can be improved to 100%. Furthermore, we also provide a constructive way to enumerate such fully efficient arrays. This implies that our method can be used in a practical synthesis system and can allow the user to derive the best fully efficient array. The clustering corresponds to the application of a quasi-linear allocation function. Indeed, the main advantage of quasi-linear allocation functions is that they provide a method for unifying clustering within the synthesis framework.

In addition, we show how the mathematical theory of clustering that we develop can be applied to *any* systolic array to derive a fully efficient systolic array. This is based on a standard view of the systolic arrays as UREs with a strongly separating hyperplane. Our methods can also be easily generalized to transform piece-wise systolic arrays [113] to fully efficient ones, by applying a *piece-wise* quasi-affine function.

Notations and Problem Definition

A systolic array consists of identical processors which are connected locally. Processors process data from input channels and send out the output through output channels to other processors at every clock cycle. To synchronize the data flow, it is possible that a processor has to be idle during some clock cycles. This leads to the concept of *extrinsic iteration interval* [98] which is defined as follows:

Definition 9.1

The *extrinsic iteration interval* of a systolic array is defined to be δ if every processor is active at exactly one out of every δ consecutive clock cycles. The *efficiency* η of the array defined as $\eta = 1/\delta$. □

For example, in the Kung-Leiserson systolic array for banded matrix multiplication [67], the processors are active once every 3 clock cycles. Therefore, $\delta = 3$ and $\eta = 1/3$.

We will also use the concept of *e-unimodular* for a matrix, which is defined in Definition 8.1 in Chapter VIII. Furthermore, we say that a vector is a normalized vector if it is e-unimodular (i.e. the gcd of all its components is 1).

Throughout this chapter, we will assume that the UREs are defined on an n-dimension integral domain $\mathcal{D}$. The standard linear transformation technique to synthesize systolic arrays from UREs can be viewed as a space-time transformation, which involves two linear transformations, namely, the timing function (represented by its norm λ) and the allocation function (represented by a normalized projection vector u or an integral $(n-1) \times n$ matrix A satisfying $Au = 0$, and u is the basis of the right null space of A). Besides the constraints stated in Chapter VI on valid timing and allocation functions, we can make the following assumptions without any loss of generality.

- λ is a normalized vector. This is because if λ is not a normalized vector, it can be written as $s\lambda'$ for some positive integer, s and normalized vector, λ'. It is easy to verify that λ' still represents a valid timing function (see [97]) and it yields a faster schedule than λ.

- The dependency graph of the computation is connected. (Otherwise, the computation consists of more than one totally independent computation, and we can rewrite them as separate UREs). This constraint is satisfied iff the dependency matrix $(D_1 \ \ldots \ D_k)$ where $D_1, \ldots, D_k$ are all dependency vectors is e-unimodular.

Geometrically, the timing and allocation functions can be unified as a single transformation from the computation domain $\mathcal{D}$ to the processor-time domain $\mathcal{T}$. This can be described by an $n \times n$ matrix $T = \begin{pmatrix} A \\ \lambda^t \end{pmatrix}$. We denote the processor space (i.e. the first $n-1$ dimensions of $\mathcal{T}$) as $\mathcal{P}$. Furthermore, we say a processor-time point $(P\ t)^t \in \mathcal{T}$ is *active* if it is the image of a computation point, i.e. there exists an $I \in \mathcal{D}$ such that $TI = (P\ t)^t$. Otherwise, $(P\ t)^t$ is said to be *inactive* (or "hole"). Intuitively, a processor-time point $(P\ t)^t$ is active iff the processor in the location P is active at time t.

Activation Patterns and Efficiency

We now establish a lemma that provides a fundamental characterization of the activation patterns of processors in arrays derived by integral linear transformations. We will also use this to show a well known "folk-theorem" regarding the efficiency of processor arrays.

The active points in a derived array can be characterized by certain properties of T. Let $C = (C_1 C_2 \ldots C_n)$ be an $n \times n$ unimodular matrix such that $AC = (E_{n-1}\ 0)$ (since A is e-unimodular, its column hermite form is $(E_{n-1}\ 0)$). Let $v^t = \lambda^t(C_1 \ldots C_{n-1})$ and $k = \lambda^t C_n$ (thus, $\lambda^t C = (v^t\ k)$). Note that since $AC = (E_{n-1}\ 0)$, we have $AC_n = 0$, i.e. C_n is a right null vector of A. Moreover, since C is unimodular, C_n must be normalized. Therefore, $C_n = \pm u$ and $k = \lambda^t C_n = \pm \lambda^t u$. The following lemma gives an important characterization of active points.

Lemma 9.1

A processor-time point $(P\ t)^t \in \mathcal{T}$ is active iff the following equation has

an integral solution j.

$$v^t P + jk = t \tag{IX.8}$$

□

Proof

Consider transformation matrix T. We have

$$T = \begin{pmatrix} A \\ \lambda^t \end{pmatrix} = \begin{pmatrix} (E_{n-1}0)C^{-1} \\ \lambda^t \end{pmatrix} = \begin{pmatrix} E_{n-1}\,0 \\ \lambda^t C \end{pmatrix} C^{-1} = \begin{pmatrix} E_{n-1} & 0 \\ v^t & k \end{pmatrix} C^{-1} \tag{IX.9}$$

Now, a processor-time point $(P\,t)^t \in \mathcal{T}$ is active iff there exists $I \in \mathcal{Z}^n$ such that

$$\begin{pmatrix} P \\ t \end{pmatrix} = TI = \begin{pmatrix} E_{n-1} & 0 \\ v^t & k \end{pmatrix} C^{-1} I$$

Let $J = C^{-1}I$. This defines a bijection $\mathcal{Z}^n \rightarrow \mathcal{Z}^n$ because C (and hence, C^{-1}) is unimodular. Hence the above system of equations has an integral solution I iff the following system has an integral solution J.

$$\begin{pmatrix} E_{n-1} & 0 \\ v^t & k \end{pmatrix} J = \begin{pmatrix} P \\ t \end{pmatrix}$$

Letting $J = (j_1 \ \ldots \ j_{n-1}\, j)^t$, we see that the solution for $(j_1 \ \ldots \ j_{n-1})^t$ is simply P. Substituting this into the last equation and simplifying, yields Eqn. IX.8. □

The following result is well known and has been reported by many researchers.

It is proved here for the sake of completeness. The details of the proof are also used later.

Remark 9.1

The extrinsic iteration interval δ, of the systolic array derived from a URE is $|\lambda^t u|$. □

Proof

We first prove that if an processor-time point $P_{t_0} = (P\, t_0)^t$ is active, then for any scalar integer α, the processor-time point $(Pt_0+\alpha k)^t$ is also active. Since $(P\, t_0)^t$ is active, there exists an integer j_0 such that

$$v^t P + j_0 k = t_0$$

Adding αk to both sides,

$$v^t P + (j_0 + \alpha)k = t_0 + \alpha k$$

Hence, processor P is active at $t_0+\alpha k$, and the extrinsic iteration interval, δ, must be a factor of k, i.e. $\delta|k$. To show that $|k|$ is exactly δ, we now prove that if processor P is active at t_1 and t_2, then $(t_1 - t_2)$ must be a multiple of k. Indeed, based on Lemma 9.1, there exist integers j_1 and j_2 such that

$$v^t P + j_1 k = t_1$$

and

$$v^t P + j_2 k = t_2$$

Hence,

$$(j_1 - j_2)k = t_1 - t_2$$

and therefore, the extrinsic iteration interval, δ, is $|k| = |\lambda^t u|$. □

Corollary 9.1

From Eqn IX.9, we further have $\det T = \pm\delta$. □

The "holes" in the processor-time space occur when T is not bijective. This is true only when T is not a unimodular matrix. The above corollary conforms to this. Theoretically, it is always possible to select an integral vector u such that $\lambda^t u = \pm 1$. This is due to the following lemma.

Lemma 9.2

For any normalized n-dimensional vector w^t, it is always possible to choose an $(n-1) \times n$ integral matrix M such that the matrix $(w\ M^t)^t$ is unimodular. □

Proof

Since $w^t = (w_1 \ \ldots \ w_n)$ is a normalized vector, there always exists an integral vector u such that $w^t u = \gcd(w_1 \ldots w_n) = 1$ and correspondingly we can choose an e-unimodular matrix M such that $Mu = 0$. It is easy to see that Eqn. IX.9 holds for M too. Therefore, $|M| = \pm w^t u = \pm$, and hence M is unimodular. □

In practice, however, there may be other factors which prohibit the choice of such allocation functions. The first factor is when the computation domain $\mathcal{D}$ is infinite (in this case, *calD* has only one extremal ray). In this situation, there is only

one projection vector (the ray of $\mathcal{D}$) which yields a finite array, as shown by Quinton [93] and this projection vector may not yield fully efficient array (i.e. $\delta = 1$). Consider banded matrix multiplication (for possibly infinitely large matrices), the computation domain is infinite and the only valid array derived by the above transformation is Kung-Leiserson array whose efficiency is only 1/3.

The second factor is due to the fact that projection vectors which yield efficient arrays do not necessarily use the least number of processors. As an example, for banded (but bounded) matrix multiplication, one projection vector yields a fully efficient array with n^2 processors while the Kung-Leiserson array uses only $w_1 w_2$ (w_1, w_2 are the bandwidths of the matrices) processors. Even though this array is not fully efficient, it may still be preferable to the other array. As a result we must investigate methods for improving the efficiency of arrays.

Quasi-Linear Allocation Functions

Based on the above discussion, we see that it is inherently restrictive to try to derive efficient arrays with only integral linear transformations. In this section we will therefore propose a new class of allocation functions and investigate its properties. This class is a strict superset of the usual integral linear allocation functions, and consists of *rational* linear functions. But to guarantee that the transformation yields a processor array, we have to change the locations (labels) of the processors denoted by rationals to integers. One of the possibilities is to use the floor function.

Definition 9.2

A *quasi-linear allocation function*, specified by an $(n-1) \times n$ full row rank rational matrix Q maps the computation at index point I to a processor

labeled* $\lfloor QI \rfloor$ □

Quasi-linear functions were proposed by Quintion [93] as a class of timing functions in systolic array design, and Van Dongen has also studied the derivation of *quasi* regular arrays (an extension of systolic arrays) using what he calls quasi-affine mappings*.

We know that Q can be decomposed as $Q = \frac{1}{d}T$ where d is the least common multiple of denominators of all the entries of Q and T is an $(n-1) \times n$ integral matrix. Moreover, T can be decomposed into form $T = MA_1$ for some $(n-1) \times (n-1)$ nonsingular integral matrix M and $(n-1) \times n$ e-unimodular integral matrix A_1 (see Lemma 5 in Chapter VIII). We can decompose M into its Smith normal form S (i.e., $M = USR$ for some unimodular U and R). Therefore, $Q = \frac{1}{d}USRA_1$. Taking $A = RA_1$ and noting that A is also e-unimodular, we can assume without loss of generality that $Q = \frac{1}{d}USA$ for some integer d, a unimodular matrix U, a diagonal D and an e-unimodular A.

Thus, any quasi-linear allocation function can be interpreted as follows. We first apply an integral projection that maps the n-dimensional index space to an $(n-1)$-dimensional integer processor space. Then we "scale" the coordinates of the processor space (multiplication by diagonal matrix S). Next we perform a basis transformation (unimodular matrix U), and then cluster all the processors within all cubes of size d^{n-1} (division by d and taking the floor). We denote by $C(P)$, the transformation $\lfloor \frac{1}{d}USP \rfloor$. Thus the quasi-linear allocation function is the composition of the projection A and

---

*The floor, $\lfloor X \rfloor$ of a rational vector $X = [x_1 \ldots x_n]$ is $[\lfloor x_1 \rfloor \ldots \lfloor x_n \rfloor]$.

*The term quasi-affine has been used somewhat differently in the literature. Quinton [93] defines quasi-affine functions as the floor of rational affine functions, and semi-linear functions as linear functions modulo a constant. Van Dongen's quasi-affine mappings involve both the floor and modulo operations.

C.

Lemma 9.3

If Q is timing conflict-free, then A should be also timing conflict-free. □

Proof

Let Q be conflict-free while A is not timing conflict-free. Then there are at least two points I_1, I_2 such that $\lambda^t I_1 = \lambda^t I_2$ and $AI_1 = AI_2$. Therefore, $\lfloor \frac{1}{d} USAI_1 \rfloor = \lfloor \frac{1}{d} USAI_2 \rfloor$, which means Q is not timing conflict-free too. This is a contradiction. □

Hence in our geometrical interpretation, A represents a valid conventional linear integer allocation function. Note that the above Lemma only gives us a necessary condition. In addition to being free from timing conflicts, any valid allocation function should also satisfy the dense-array constraint. Moreover, we want to ensure that the the resultant arrray is spatially fully regular (i.e., all the processors are identical and have identical connections). We shall see that, unlike in the case of integer allocation functions, any candidate function that satisfies the other constraints may not necessarily guarantee spatial regularity.

In the following, we will show that if there is no basis transformation (U is E_{n-1}), then any valid quasi-linear allocation function that yields a fully regular dense array must be a "clustering" of an array derived by A. We will henceforth assume that $Q = \frac{1}{d} SA = DA$, where $D = diag(\frac{p_1}{q_1}, \ldots, \frac{p_{n-1}}{q_{n-1}})$ where p_i, q_i are all integers and for each i, p_i and q_i are relatively prime. First, we prove a technical lemma.

Lemma 9.4

Let i, p, and q be integers, $q, p \neq 0$. The number of distinct j's that

satisfy the equation

$$i = \lfloor \frac{pj}{q} \rfloor \qquad \text{(IX.10)}$$

is $\lceil \frac{q(i+1)}{p} \rceil - \lceil \frac{qi}{p} \rceil$. Moreover, Eqn. IX.10 has at least one integral solution j for any i iff $p \leq q$. □

Proof

j is a valid solution to Eqn. IX.10 iff $i \leq \frac{pj}{q} < i+1$, i.e., $\frac{qi}{p} \leq j < \frac{q(i+1)}{p}$.

Suppose j_s is the smallest solution and j_b is the biggest solution, we know the number of distinct integral solutions should be $j_b - j_s + 1$. But it is obvious that $j_s = \lceil \frac{qi}{p} \rceil$ and $j_b + 1 = \lceil \frac{q(i+1)}{p} \rceil$. Therefore, there are exactly $\lceil \frac{q(i+1)}{p} \rceil - \lceil \frac{qi}{p} \rceil$ integral solution j's.

If $p > q$, for $i = 0$, $\lceil \frac{q(i+1)}{p} \rceil = \lceil \frac{qi}{p} \rceil = 0$. Therefore, Eqn. IX.10 does not have an integral solution. Conversely, if $p \leq q$, we have $\frac{q}{p} \geq 1$ and so $\lceil \frac{q(i+1)}{p} \rceil \geq \lceil \frac{qi}{p} \rceil + 1$. Therefore, Eqn. IX.10 has at least one integral solution. □

Lemma 9.5

The quasi-linear allocation function specified by Q yields a dense array iff $p_m \leq q_m$ for all $m = 1, \ldots, n-1$. □

Proof

The array derived by Q is dense iff for any $J = (j_1 \ldots j_{n-1})^t \in \mathcal{Z}^{n-1}$, there always exists $J \in \mathcal{Z}^n$ such that $J = \lfloor QJ \rfloor = \lfloor DAI \rfloor$. Since A is e-unimodular, there always exists an integral solution I to $J' = AI$ for any integral vector J'. Therefore, we need to find the conditions under which the equation $J = \lfloor DI \rfloor$ has integral solution $I \in \mathcal{Z}^{n-1}$ for *any*

integral $J \in \mathcal{Z}^{n-1}$. Since D is diagonal $J = \lfloor DI \rfloor$ is equivalent to $n-1$ independent equations of the form

$$j_m = \lfloor \frac{p_m i_m}{q_m} \rfloor \qquad \text{(IX.11)}$$

for $m = 1, \ldots, n-1$. By Lemma 9.4, each one of them has an integral solution j_m for any integral i_m iff $p_m \leq q_m$. □

We thus see that $\mathcal{C}$ is a "clustering" of processors, i.e., a surjective map on $\mathcal{Z}^{n-1}$. Furthermore we desire that the clustered final array consists of identical processors, which implies that an equal *number* of processors are mapped to each cluster (otherwise some clusters will be different from others). The following considerations enable us to impose additional constraints on D. $\mathcal{C}$ maps $J = (j_1 \ldots j_{n-1})^T$ to $I = (i_1 \ldots i_{n-1})^T$ iff Eqn. IX.11 holds for $m = 1, \ldots, n-1$. From Lemma 9.4 we know that for each m, the number of distinct solutions to Eqn. IX.11 is $N_m = \lceil \frac{q_m(i_m+1)}{p_m} \rceil - \lceil \frac{q_m i_m}{p_m} \rceil$. We desire that this must be constant, independent of i_m. Now let $q_m = kp_m + r$ such that $0 \leq r < p_m$ and $i_m = lp_m + i_0$ where $0 \leq i_0 < p_m$. By simple arithmetic manipulation we can show that $N_m = k + \lceil \frac{r(i_0+1)}{p_m} \rceil - \lceil \frac{ri_0}{p_m} \rceil$. It is easy to see that, if $r > 0$, N_m is not a constant because $N_m = k+1$ for $i_0 = 0$ and $N_m = k$ for $i_0 = p_m - 1$. For example, let $q_m = 11, p_m = 8$, we have $k = 1$ and $r = 3$. $N_m = 2, 1, 2, 1, 1, 2, 1, 1$, for $i_m = 0, 1, \ldots, 7$ and it repeats the pattern again for other i_m. Hence, if we are to derive fully regular arrays with our quasi-linear allocation functions, r must be zero, i.e., q_m must be divisible by p_m. Thus. we have the following

Lemma 9.6

Q results in a fully regular array iff for all $m = 1, \ldots n-1$, $p_m = 1$, i.e.,

$D = diag(\frac{1}{q_1}, \ldots, \frac{1}{q_{n-1}})$. □

Proof

Obvious from the above discussion. □

We are thus able to characterize quasi-linear functions when they are to be used as allocation functions in systolic array design. They must be the composition of some integral linear allocation function, A, and a clustering operation $\mathcal{C} = \lfloor D \rfloor$, where $D = diag(\frac{1}{d_1} \ldots \frac{1}{d_{n-1}})$. $\mathcal{C}$ thus merges all processors whose labels in the m-th coordinate range from kd_m to $(k+1)d_m - 1$. Thus all the processors within a rectangular parallelepiped are clustered together into a single processor (called rectangular clustering). Note that Lemma 9.3 gives us necessary, but not sufficient conditions for a quasi-linear allocation function to be timing conflict-free. To obtain a better characterization, one must show that in the array derived by A, among all the processors within a rectangular parallelepiped, there is never a time instant when more than one processors are active simultaneously. In this paper, we are concerned with a specific motivation, namely the derivation of efficient arrays. For this, we are interested in a "maximal" clustering (one where we cluster as many processors as possible).

Synthesizing Fully Efficient Systolic Arrays

From the previous section, we see that any quasi-linear allocation function is a rectangular clustering of the processors derived by some integral linear allocation function. It is natural to ask the question whether such allocation functions are general enough, i.e., can they describe *all* possible clusterings of arrays derived by *any* linear allocation function. To answer this, we need to return to the motivation

for clustering, namely to improve the efficiency of the derived arrays. We will show that there always exist quasi-linear allocation functions which yield arrays with 100% efficiency (i.e., there are δ processors in each cluster).

A standard method of clustering is to merge all the processors within a (not necessarily rectangular) parallelepiped . This corresponds to a basis transformation followed by rectangular clustering. We shall describe how to perform this basis transformation. Since such a basis transformation merely renames the processor labels, the composition of this and the original linear allocation function is a valid allocation function that yields the same array. Thus, quasi-linear allocation functions can describe arbitrary parallelepiped clusterings of *any* array derived by a linear allocation function.

Thus, given an array derived by a linear allocation function, we are interested in determining parallelepiped regions containing exactly δ processors that satisfy the following property: at any time instant, exactly one of the processors is active (i.e., the image of a point in the original computation domain). We shall first study the activation patterns along parallel lines in the processor space, and then generalize these results to parallelepipeds.

Activation Patterns of Processors along Parallel Lines

For a given t, the space $\{(P\ t)^t \mid P \in \mathcal{P}\}$ is called a *snapshot* at time t. Given an $(n-1)$-dimensional normalized vector μ, a processor point P and a time instant t, $L(\mu, P, t)$ denotes a line, $\{(P + \alpha\mu\ \ t)^t \mid \alpha \in \mathcal{Z}\}$ in the snapshot at time t. The set, $\{L(\mu, P, t) \mid P \in \mathcal{P}, t \geq 0\}$ of all parallel lines in all snapshots is denoted by $F(\mu)$.

Lemma 9.7

For any line $L(\mu, P, t)$, if there is an active point in it, then there are infinitely many active points in it. Such a line is called an active line. □

Proof

Suppose that $(P + \alpha_0\mu \;\; t)^t$ is an active point in line $L(\mu, P, t)$, based on Lemma 9.1, there exists an integer j_0 such that

$$v^t(P + \alpha_0\mu) + kj_0 = t. \tag{IX.12}$$

It is easy to see that $Q = (P + (\alpha_0 + \alpha k) \;\; t)^t$ (α is an arbitrary integer) in line $L(\mu, P, t)$ is active too. This is because we can let $j = j_0 - \alpha v^t\mu$, j and Q satisfy Eqn. IX.8. Therefore, there are infinitely many active points in line $L(\mu, P, t)$. □

Lemma 9.8

Active points in any active line occur periodically and periods of all active lines in any family $F(\mu)$ are the same. □

Proof

We know that line $L(\mu, 0, 0)$ is active because 0 is an active point at time 0. Let $Q_0 = (\alpha\mu \;\; 0)$ ($\alpha \neq 0$) be the closest active point to 0 in line $L(\mu, 0, 0)$ (based on Lemma 9.7, we know Q_0 always exists). The number of holes between 0 and Q_0 is $(|\alpha| - 1)$. Further, let $Q_1 = (P + \alpha_1\mu \;\; t)^t$ and $Q_2 = (P + \alpha_2\mu \;\; t)^t$ be two consecutive active points in line $L(\mu, P, t)$; the number of holes between them is $(|\alpha_1 - \alpha_2| - 1)$. We prove that this is also equal to $(|\alpha| - 1)$, which will prove the lemma.

Since Q_0, Q_1 and Q_2 are active, there exist integral points I_0, I_1 and I_2 such that $TI_0 = Q_0$, $TI_1 = Q_1$ and $TI_2 = Q_2$. Now, $T(I_1 - I_2) = Q_1 - Q_2 = ((\alpha_1 - \alpha_2)\mu \;\; 0)^t$, which is an active point in line $L(\mu, 0, 0)$. The number of integral points between it and 0, which is $(|\alpha_1 - \alpha_2| - 1)$, should be at least $(|\alpha| - 1)$ (since Q_0 is closest). Thus $|\alpha| \leq |\alpha_1 - \alpha_2|$.

Conversely, $T(I_0 \pm I_1) = Q_0 \pm Q_1 = (P + (\alpha \pm \alpha_1)\mu \;\; t)^t$ is also an active point in line $L(\mu, P, t)$. The number of integral points between Q_1 and $Q_0 \pm Q_1$ is $|(\alpha_1 \pm \alpha) - \alpha_1| = |\alpha|$ which can be no less than $(|\alpha_1 - \alpha_2| - 1)$, the number of holes between Q_1 and Q_2. Therefore, $|\alpha_1 - \alpha_2| \leq |\alpha|$. □

We call α the period of the active line. The following theorem further clarifies the above results.

Theorem 9.1

The period of any active line $L(\mu, P, t)$ is a factor of δ. □

Proof

As in Lemma 9.8, let Q_0 be the closest active point to 0 in line $L(\mu, 0, 0)$. The period of the line is the smallest (non-zero, positive) value of α such that $(\alpha\mu \;\; 0)^t$ is an active point. From Lemma 9.1, this is given by the smallest value of α for which Eqn IX.13 below has an integral solution j.

$$\alpha v^t \mu + kj = 0 \qquad \text{(IX.13)}$$

By elementary number theory, $\frac{|k|}{\gcd(k, v^t\mu)}$ is the smallest positive α which satisfies Eqn. IX.13, and hence $\alpha \mid k$, i.e., $\alpha \mid \delta$. □

Example 9.1

Suppose $v^t = (1\ 3), k = 6$. First consider $\mu = (3, -1)^t$, we have $v^t\mu = 0$. Hence, in family $F(\mu)$, every point in line $L(\mu, 0, 0)$ is active. But in line $L(\mu, 0, 1)$, every point is inactive. On the other hand, if we let $\mu = (5\ -1)$, then $v^t\mu = 2$. The periods of any active line in family $F(\mu)$ are $6/2 = 3$. Notice even in this family, there are still some inactive lines. For example, $L(\mu, 0, 1)$ is inactive. □

Clustering of processors in a Parallelepiped

In order to cluster the processors in a parallelepiped, we first need $(n-1)$ directions $\nu_1, \ldots, \nu_{n-1}$ along which the parallelepiped is formed. We are only interested in parallelepipeds of volume δ, so we factorize δ into $\delta = \delta_1 \ldots \delta_{n-1}$, and these factors together with the basis vectors define our family of parallelepipeds.

Definition 9.3

A factorization $\delta = \delta_1\delta_2 \ldots \delta_{n-1}$ of δ, a matrix N of $(n-1)$ basis vectors $(N = (\nu_1, \ldots, \nu_{n-1})$, for the processor space $\mathcal{P}$, and a processor-time point $(Pt)^t$ define a *parallelepiped*, denoted by $\Pr(N, P, t)$ of volume δ as follows.

$$\Pr(N, P, t) = \{(P + N(l_1 \ldots l_{n-1})^t\ t)^t \mid 0 \leq l_i < \delta_i, i = 1, .., n-1\}$$

□

It should be noted that the requirement that $\nu_1, \nu_2, \ldots, \nu_{n-1}$ is a basis for the processor space $\mathcal{P}$ is necessary. Otherwise, there may be some integral points in the parallelepiped which cannot be written as a linear combination of the ν_i's, and the

number of integral points in the parallelepiped may be greater than δ. Our main result is that such clustering into parallelepipeds is always possible, provided that the factors of δ are mutually co-prime.

Recall that $\lambda^t C = (v^t\ k)$. Since v^t is not necessarily normalized, let g be the gcd of all components of v^t, so $v^t = gw^t$ for some normalized w^t. Now, $\lambda^t C = (v^t\ k)$. Since λ^t is normalized and C is unimodular, $(v^t\ k)$ is also normalized, i.e., $\gcd(v_1, \ldots v_{n-1}, k) = \gcd(g, k) = 1$.

Our intuition in choosing the parallelepipeds is to choose the $n-1$ basis vectors, ν_i's such that the periods of the active lines in the families $F(\nu_i)$ are δ_i respectively. From Theorem 9.1, each ν_i must satisfy $\delta = \delta_i \gcd(\delta, v^t\nu_i)$. So $\delta = \delta_i \gcd(\delta, gw^t\nu_i) = \delta_i \gcd(\delta, w^t\nu_i)$ (since δ and g are coprime as shown above). Denoting $\frac{\delta}{\delta_i}$ by k_i, we have, $w^t\nu_i = c_i k_i$ for some c_i where $\gcd(\delta, c_i) = 1$. Thus $v^t\nu_i = gc_i k_i$, and each ν_i must be a solution of the i-th equation in the following system of diophantine equations.

$$\begin{aligned} v^t x_1 &= gc_1 k_1 \\ v^t x_2 &= gc_2 k_2 \\ &\vdots \\ v^t x_{n-1} &= gc_{n-1} k_{n-1} \end{aligned}$$

As shown by Banerjee ([6], Th. 5.4.2, p.81), the general solution to the i-th equation is given by

$$x_i = U^t \begin{pmatrix} c_i k_i \\ t_1^i \\ \vdots \\ t_{n-2}^i \end{pmatrix}$$

where t_j^i for $j = 1, \ldots, n-2$ are arbitrary integers and U is any unimodular matrix that satisfies $Uv = (g, 0, \ldots, 0)^t$. Such a U can always be found, and thus, our ν_i's satisfy the desired constraint iff the matrix

$$V = \begin{pmatrix} c_1 k_1 & c_2 k_2 & \ldots & c_{n-1} k_{n-1} \\ t_1^1 & t_1^2 & \ldots & t_1^{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ t_{n-2}^1 & t_{n-2}^2 & \ldots & t_{n-2}^{n-1} \end{pmatrix}$$

is unimodular because $N = (\nu_1 \ \ldots \ \nu_{n-1}) = UV$. The following lemma gives us necessary and sufficient conditions for this.

Lemma 9.9

V is unimodular iff $\gcd(\delta_i, \delta_j) = 1$ for all $i \neq j$ and $\gcd(c_1, \ldots, c_{n-1}) = 1$.

□

Proof

If part. Let, $\gcd(\delta_i, \delta_j) = 1$, for all $i \neq j$ and $\gcd(c_1, \ldots, c_{n-1}) = 1$. First, we prove that $(k_1 \ \ldots \ k_{n-1})$ is a normalized vector. Suppose p is a common prime divisor for $k_1 \ \ldots \ k_{n-1}$. Obviously, $p|k_1$. There exists $i \neq 1$ such that $p|\delta_i$, since $k_i = \delta_1 \ldots \delta_{i-1}\delta_{i+1} \ldots \delta_{n-1}$. Similarly, since $p|k_i$, there exists $j \neq i$ such that $p|\delta_j$. Because $\gcd(\delta_i, \delta_j) = 1$, p must be 1. Therefore, $(k_1, \ldots, k_{n-1})$ is normalized. We now prove that $(c_1 k_1 \ \ldots \ c_{n-1} k_{k-1})$ is also normalized. Again, suppose p is a common prime divisor for all $c_i k_i$, then p either divides c_i or k_i. Since $\gcd(c_1, \ldots, c_{n-1}) = 1$ and $\gcd(k_1, \ldots, k_{n-1}) = 1$, there exist i, j $(i \neq j)$ such that $p|c_i, p|k_j$. But from $\gcd(c_i, \delta) = 1$, we

know $\gcd(c_i, k_j) = 1$. Hence, $p = 1$. By Lemma 9.2, we know it is always possible to choose t_i^j's such that V is unimodular.

Only If Part. Clearly, $\gcd(c_1, \ldots, c_{n-1}) = 1$ (otherwise the first row of V is not normalized, and hence V cannot be unimodular). Without loss of generality, suppose $\gcd(\delta_1, \delta_2) = d \neq 1$. $d|k_1$ because $\delta_2|k_1$. Also, $d|k_i$ for $i \neq 1$ because $\delta_1|k_i$ for all $i \neq 1$. Therefore, we know $d|c_i k_i$ for all $i = 1, \ldots, n-1$. Hence, V is not unimodular. □

To recap the discussion so far, we have the following constraints in picking the basis vectors for our parallelepipeds:

1. Factorize δ into $\delta = \delta_1 \ldots \delta_{n-1}$ such that the factors are pairwise mutually coprime.

2. Choose $c_1 \ldots c_{n-1}$ which satisfy $\gcd(c_i, \delta) = \gcd(c_1, \ldots, c_{n-1}) = 1$.

These can always be satisfied. In particular, we can let $\delta_1 = \delta$, and $\delta_2 = \ldots = \delta_{n-1} = 1$, and $c_1 = \ldots = c_{n-1} = 1$. In this case the parallelepiped degenerates into a line. In practice, we can use the prime factorization of δ to systematically enumerate all possible choices of the δ_i's. Similarly, there are many additional choices for c_i's. Practical considerations can also be used to guide the selection. We now prove that any parallelepiped of any snapshot formed by the basis vectors as chosen above contains exactly one active point.

Lemma 9.10

For any time instant t, there is exactly one active point in any parallelepiped $\Pr(N, P, t)$. □

Proof

First, we prove that there is at most one active point in $\Pr(N, P, t)$. Suppose there are two active points, say, $(P_1\ t)^t$ and $(P_2\ t)^t$ in this parallelepiped. It is easy to see that the point $(P_1\ t)^t - (P_2\ t)^t$ is also active. This means that there are integers $l_1, l_2, \ldots, l_{n-1}$ where $0 \leq l_i < \delta_i$ for $i = 1, \ldots, n-1$ such that $\begin{pmatrix} l_1\nu_1 + \ldots + l_{n-1}\nu_{n-1} \\ 0 \end{pmatrix}$ is active. From Lemma 9.1, we know that there is an integral solution to the Eqn. IX.8, i.e. there is an integer J_n such that

$$l_1 v^t \nu_1 + \ldots + l_{n-1} v^t \nu_{n-1} + J_n k = 0$$

This can be further simplified to

$$g(l_1 c_1 k_1 + \ldots + l_{n-1} c_{n-1} k_{n-1}) + J_n k = 0$$

Notice that for any $i \neq j$, $\delta_i | k_j$. Therefore, dividing both sides of the above equation by δ_i, we have $\delta_i | g l_i c_i k_i$. Because $\gcd(g, \delta) = 1$, hence $\gcd(g, \delta_i) = 1$. Also, $\gcd(c_i, \delta) = 1$, and hence $\gcd(c_i, \delta_i) = 1$. Furthermore, $\gcd(\delta_i, \delta_j) = 1$ (for $i \neq j$), and hence $\gcd(\delta_i, k_i) = 1$. So $\delta_i | l_i$. Because $0 \leq l_i < \delta_i$, l_i must be 0, i.e. P_t^1 is the same as P_t^2.

To prove that there is at least one active point in the parallelepiped, again, consider Eqn. IX.8 which can be simplified to

$$g(l_1 c_1 k_1 + \ldots + l_{n-1} c_{n-1} k_{n-1}) + J_n k = t - v^t P \qquad \text{(IX.14)}$$

We thus want to prove that there are integers $l_1, \ldots, l_{n-1}$ for $0 \leq l_i < \delta_i$ and some integer J_n as the solution to the above equation. To prove this, we will first prove $\gcd(gc_1k_1, \ldots, gc_{n-1}k_{n-1}, k) = 1$. Consider any prime common divisor p of these integers, we prove $p = 1$. Because $\gcd(g, k) = 1$, if $p|g$, then we already prove that $p = 1$ (because $p|k$ too). If p is not a factor of g, we have $p|c_ik_i$ for every $i = 1, \ldots, n-1$. Because $\gcd(c_1k_1, \ldots, c_{n-1}k_{n-1}) = 1$, p must be 1.

Therefore, there exist integers $l'_1, \ldots, l'_{n-1}, J'_n$ such that

$$l'_1gc_1k_1 + \ldots + l'_{n-1}gc_{n-1}k_{n-1} + J'_nk = 1$$

Denoting $t - v^tP$ as t' and multiplying both sides of the equation by t', we have

$$t'l'_1gc_1k_1 + \ldots + t'l'_{n-1}gc_{n-1}k_{n-1} + J'_nt'k = t'$$

For $t'l'_i$, we can always find two integers q_i and l_i where $0 \leq l_i, \delta_i$ such that $t'l'_i = q_i\delta_i + l_i$ (i.e. l_i is the remainder of dividing $t'l_i$ by δ_i). Notice that $\delta_ik_i = \delta = \pm k$. The left hand side of the above equation becomes

$$(\sum_{i=1}^{n-1} l_igc_ik_i) + J'_nt'k + q_1gc_1\delta_1k_1 + \ldots + q_{n-1}gc_{n-1}\delta_{n-1}k_{n-1}$$

$$= (\sum_{i=1}^{n-1} l_igc_ik_i) + (J'_nt' \pm (q_1gc_1 + \ldots + q_{n-1}gc_{n-1}))k$$

Let $J_n = (J'_nt' \pm (q_1gc_1 + \ldots + q_{n-1}gc_{n-1}))$, l_i and J_n together satisfy Eqn. IX.14. □

Therefore, We can merge all the δ processors in such parallelepipeds into one processor. The new processor does not need to have extra processing function units (but may need some additional links and registers) and it will be active in the whole computation. This leads to the following theorem.

Theorem 9.2

> It is always possible to merge δ neighboring processors which form a parallelepiped in the processor space to derive a fully efficient array. The new array has the same computation time as the original one and has the same cost except for some additional links and registers. □

The quasi-linear allocation function that corresponds to the above clustering scheme is given by $P = \lfloor DNAI \rfloor$, where D is an $(n-1) \times (n-1)$ diagonal matrix, $D = diag(1/\delta_1, \ldots, 1/\delta_{n-1})$. Such a clustering will save on the number of functional units in the final array by a factor of δ, at the expense of some complexity in the number of registers, the interconnections between the processors, and the control structure of the processor. There are many factors that will determine the final processor cost, as discussed below.

In our analysis so far, we have not considered the actual domain of computation. Because the domain may be an arbitrary convex polyhedron, the parallelepipeds chosen may not completely tile the whole processor domain. In this case, some of the boundary processors of the final array may not operate at 100% efficiency. However, the *number* of such processors is at least an order of magnitude smaller than the total number of processors in the array. The effect of this can be further mitigated by appropriately choosing the shape and size of the parallelepiped, i.e., the factorization of δ and the matrix V of Lemma 9.9.

As mentioned above, the clustering approach will introduce additional registers and links. The problem of reducing the additional cost imposed by a clustering has been addressed by Van Dongen [117], Bu and Deprettere [18], and other researchers. The degree of freedom that we have can be exploited to systematically investigate the design choices in these methods. In particular we make the following observations. Additional registers may be necessary if certain communications are "internalized". Also, the total amount of communication is fixed, so these registers are "compensated" by reduced inter-processor communication. If we want to minimize the inter-processor communication (i.e., maximize the number of dependencies that are internalized), note that processor $\lfloor DNA(I+d) \rfloor = \lfloor DNAI + DNAd \rfloor$ needs a data from processor $\lfloor DNAI \rfloor$. It is easy to see that this can be achieved if we minimize the absolute value of each entry in vector $DNAd$. This implies that, for each dependency vector d, we should minimize the absolute value of components of vector NAd, which is determined by the base transformation matrix N.

Another factor that must be considered is that, depending on the "shape" of the parallelepipeds, the same data dependency in the original problem domain may be mapped to two or more different links, which are active at different time instants. This results in reduced hardware utilization. A simple heuristic that can often alleviate this problem is to choose as many of the data dependency vectors as possible to serve as the bases of the parallelepiped.

Optimal Clustering of Arbitrary Systolic Arrays

So far, we have addressed the problem of deriving efficient systolic arrays in the context of synthesis. There are, however, many systolic arrays which are not derived from UREs by the conventional linear transformation. To transform such arrays into

efficient ones, we study how to apply our theory to an arbitrary systolic array. First, let us recall the standard points of view of a systolic array.

By Rao and Kailath [99], a systolic array implements an RIA . This RIA is defined in a processor-time space. More precisely, suppose the processor space is defined in $\mathcal{P} \subset \mathcal{Z}^{n-1}$, then the RIA is defined in $\mathcal{Z}^n$ as follows:

If there is a link with a delay D_t ($D_t \geq 1$) from processor p_1 to processor $p_1 + D$ in the processor space, then for any time t, processor-time point $(p_1 + D\ t + D^t)^t$ depends on $(p_1\ t)^t$. Hence, there is a uniform dependency $\begin{pmatrix} D \\ D_t \end{pmatrix}$.

Generally, if $D_1, \ldots, D_k$ (we assume $k \geq n$, otherwise, the RIA can be transformed into a lower dimension space) are all the link vectors in the processor space and $D_1^t, \ldots, D_k^t$ are the time delays along the i-th link respectively, then the RIA implemented by the array is defined by the matrix D formed by the dependency vectors as $\begin{pmatrix} D_1 & \ldots & D_k \\ D_1^t & \ldots & D_k^t \end{pmatrix}$.

It might seem straightforward to use the above technique because the array is derived by a projection of the RIA along the time axis. But if the array is not 100% efficient, then the computation dag of this RIA consists of k *disconnected* components in $\mathcal{Z}^n$, which violates the assumption we made on page 193. Hence our previous results cannot be applied directly.

Let us examine processor-time points in $\mathcal{Z}^n$. Some of these points correspond to useful computations (i.e. the processors are active) and others do not. Moreover, if the extrinsic iteration interval is δ, then along the time axis, there is an active processor-time point every δ points.

Basically, if we assume that the origin (i.e. $0 \in \mathcal{Z}^n$) is active, then any active

point will connected to the origin in the dag of the RIA. Thus, a processor-time point $(P\ t)^t$ is active iff it can be represented by a linear combination of the dependency vectors of the dag. This leads to the following proposition.

Remark 9.2

A processor-time point $(P\ t)^t$ is active iff there exists a k-dimensional integral vector J such that $DJ = (P\ t)^t$. □

Let $D = \begin{pmatrix} D_p \\ D_t \end{pmatrix}$ where $D_p = (D_1\ \ldots\ D_k)$ and $D_t = (D_1^t\ \ldots\ D_k^t)$. D_p is the $n-1 \times k$ connection matrix for the processor space. To guarantee that the array is dense (i.e. every integral point in $\mathcal{P}$ is a valid processor), D_p must be e-unimodular (Lemma 4, in [128]). Moreover, it is reasonable to assume that D_t is a normalized vector because, otherwise, we can get a faster array by just simply replacing D_t with D_t' where $D_t = cD_t'$ and D_t' is normalized. The matrix D is thus analogous to the transformation matrix T that we have studied so far, except that it is not square.

Because D_p is e-unimodular, there exists a $k \times k$ unimodular matrix $U = (U_1\ \ldots\ U_k)$ such that $D_pU = (E_{n-1}\ 0)$. Define $w^t = D_t(U_1\ \ldots\ U_{n-1})$ and $l_i = D_tU_i$ for $i = n, \ldots, k$, thus $D_tU = (w^t\ l_n\ \ldots l_k)$. We therefore have the following analogue of Lemma 9.1 (the proof is also analogous, and omited for brevity).

Lemma 9.11

A processor-time point $(P\ t)^t \in \mathcal{T}$ is an active point iff the following equation has an integral solution $J_n, \ldots, J_k$.

$$w^tP + J_nl_n + \ldots + J_kl_k = t \qquad \text{(IX.15)}$$

□

The key difference between Eqn. IX.8 and Eqn. IX.15 is that in Eqn. IX.15, there are $l_n, \ldots, l_k$ instead of k in Eqn. IX.8. The following Lemma enables us to eliminate this difference too.

Lemma 9.12

Eqn. IX.15 has integral solution $J_n, \ldots, J_k$ iff the following equation has an integral solution J

$$w^t P + Jl = t \tag{IX.16}$$

where $l = \gcd(l_n, \ldots, l_k)$ □

Proof

Let $(l_n, \ldots, l_k) = l(l'_n, \ldots, l'_k)$ and $\gcd(l'_n, \ldots, l'_k) = 1$. It is easy to see that if Eqn. IX.15 has integral solution $J_n, \ldots, J_k$, then $J = l'_n J_n + \ldots + l'_k J_k$ is an integral solution to Eqn. IX.16.

Conversely, suppose Eqn. IX.16 has an integral solution J. Because $\gcd(l'_n, \ldots, l'_k) = 1$, there are integers $m_n, \ldots, m_k$ such that $m_n l'_n + \ldots + m_k l'_k = 1$. Therefore, $Jm_n l'_n + \ldots + Jm_k l'_k = J$ and Eqn. IX.16 becomes

$$w^t P + l(Jm_n l'_n + \ldots + Jm_k l'_k) = t \tag{IX.17}$$

Therefore, $J_n = Jm_n, \ldots, J_k = Jm_k$ is an integral solution to Eqn. IX.15.

□

We then have the following analogue (proof omited) of Remark. 1

Theorem 9.3

The extrinsic iteration interval δ of any systolic array is l. □

Since a processor-time point in the array is completely characterized by Eqn. IX.16, which is exactly the same as Eqn. IX.8, we can use the technique of Th. 9.2 to merge δ (i.e., l) neighboring processors within a parallelepiped and derive for *any* systolic array a 100% efficient array.

The Array for Algebraic Path Problem

A hexagonal array for algebraic path problem(APP) was proposed by Rote [102]. The extrinsic iteration interval δ of the array is 3. For an $n \times n$ matrix, the number of processors is $(n+1) \times (n+1) - 1$. There are seven types of processors in the array. Most of them ($(n-1) \times (n-1)$) are of the same type (type A). Fig. 46 shows an 11×11 array for a 10×10 matrix. A conventional basis is also shown in the figure.

To derive a fully efficient array, we first try to merge the subarray which consists of all type A processors. We have the following interconnection matrix:

$$D = \begin{pmatrix} D_p \\ D_l \end{pmatrix} = \begin{pmatrix} 0 & 1 & -1 \\ 1 & 0 & -1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 3 \end{pmatrix} \begin{pmatrix} 0 & 1 & -1 \\ 1 & 0 & -1 \\ 0 & 0 & 1 \end{pmatrix} \qquad \text{(IX.18)}$$

Hence, we can conclude that the extrinsic iteration interval δ for this subarray is 3, which is a prime number. All possible parallelepipeds which we can merge have a 3×1 aspect ratio (i.e., δ_1 and δ_2 are fixed at 3 and 1 respectively). Furthermore, we have $w^t = (1\ 1)$, $g = 1$. The basis transformation matrix is determined by the following two equations (each equation independently determines one basis vector).

$$x + y = c_1$$

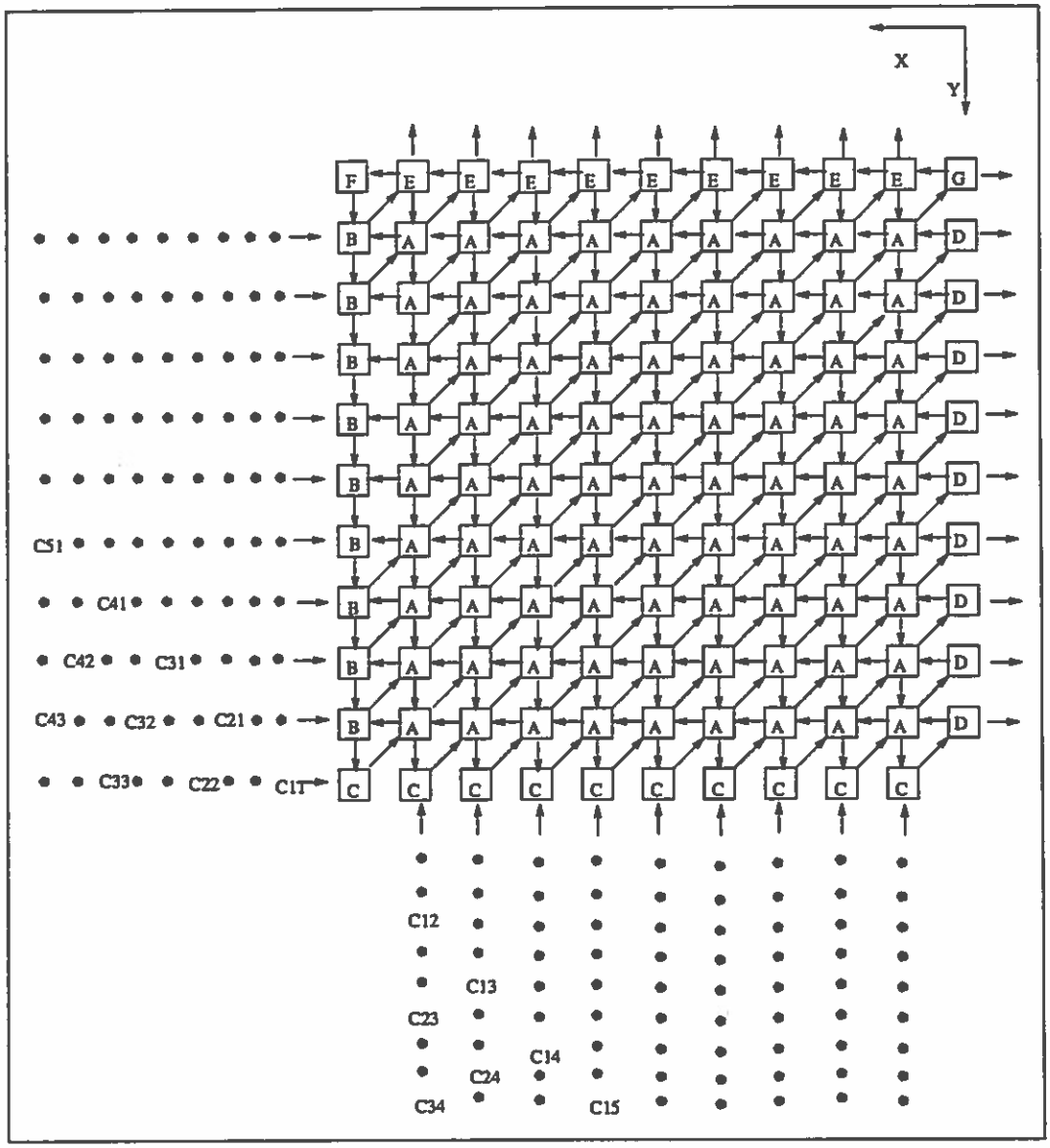


Figure 46: Rote's Hexagonal Array for 10×10 Algebraic Path Problem.

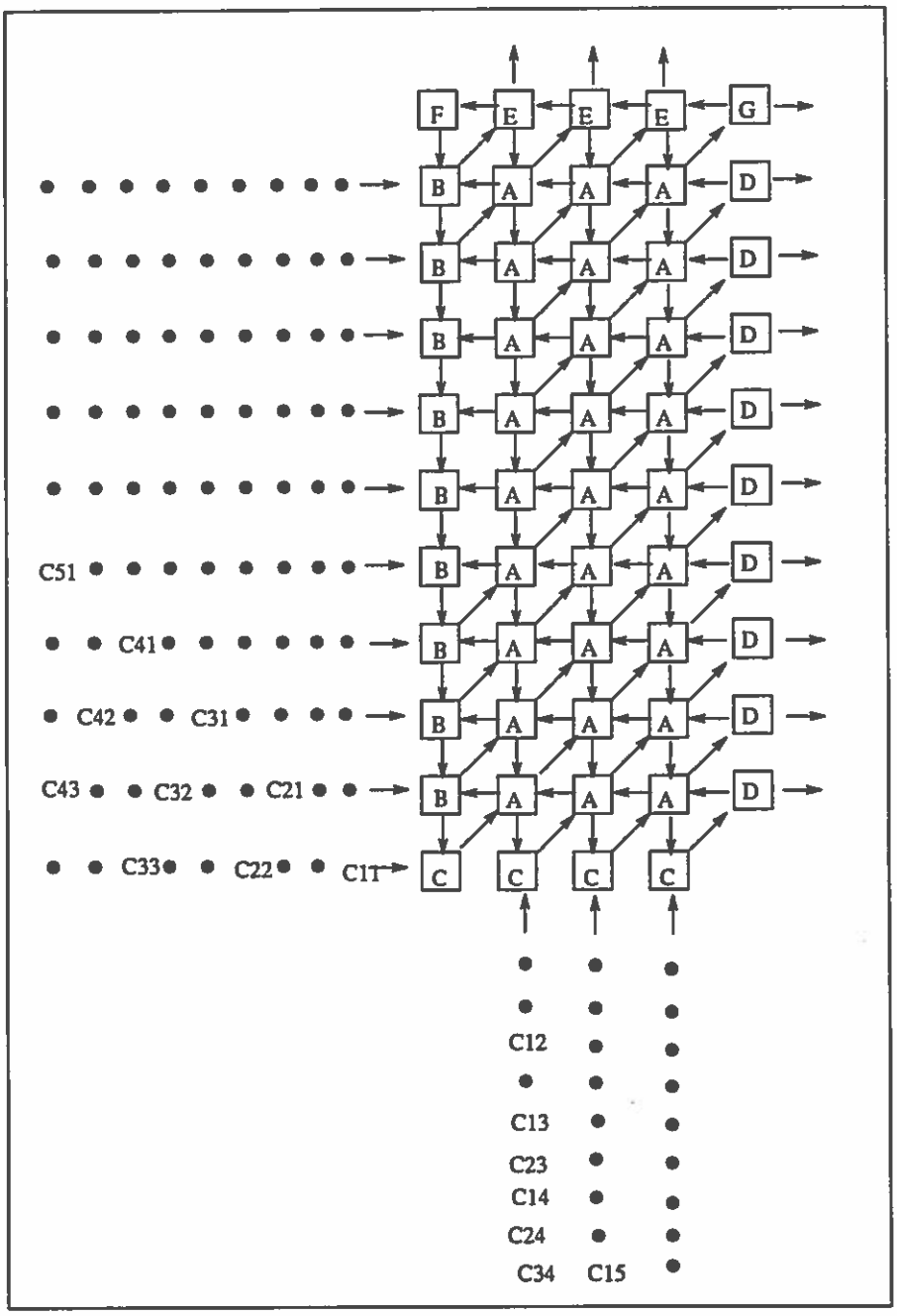


Figure 47: The Final Array for 10 × 10 APP Obtained by Merging Horizontally. A1, C1 and E1 Represent the Processors Merged from Type A, C and E Processors Respectively.

$$x + y = 3c_2$$

where $\gcd(c_1, c_2) = 1$ and $\gcd(c_1, 3) = \gcd(c_2, 3) = 1$.

There are many ways to pick the basis vectors. For example, if we choose $c_1 = c_2 = 1$, one possible set of solutions is $\nu_1 = (1\ 0)^t$ and $\nu_2 = (4\ -1)^t$. Another one is $\nu_1 = (0\ 1)^t$ and $\nu_2 = (-1\ 4)^t$. Thus, we can cluster three neighboring processors vertically or horizontally. Further, if we choose $c_1 = 2$ and $c_2 = 1$, we can get $\nu_1 = (1\ 1)^t$ and $\nu_2 = (1\ 2)^t$ as yet another solution which represents clustering along the diagonal. It should be noted that $(1\ -1)^t$ is never a solution to the first equation, because if it were, c_1 must be 0, hence $\gcd(c_1, 3) = 3$ which does not satisfy the condition as stated above. Similarly, it can never be a solution to the second equation. In fact, in the array, all processors along the direction $(1\ -1)^t$ are active or inactive at the same time.

It can be easily seen that along x-axis (or y-axis), only one out of 3 consecutive type C (or type B) processors is active at the same time unit. This is also true for type E (or type D) processors. Therefore, the best way to cluster the whole array is to cluster 3 consecutive processors along either x-axis or y-axis. Fig. 47 shows the final array derived by clustering 3 consecutive processors along x-axis. In general, the final array has $(\lceil \frac{n-1}{3} \rceil + 2) \times (n + 1) - 1$ number of processors.

Conclusions

In this chapter, we have studied the problem of extending the conventional techniques for systolic array design so that the derived arrays can be guaranteed to have 100% efficiency. We first investigated conditions for a quasi-linear function to be a valid allocation function for systolic array design. We have shown that a quasi-linear

function can be decomposed into $Q(I) = \lfloor \frac{1}{d} USA(I) \rfloor$ and proved that if U is identity, such a function is no more than a parallelepiped clustering. We then proceeded to prove that by clustering, it is always possible to improve a systolic array into a fully efficient one. This is achieved without affecting any of the other properties such as throughput, computation time, I/O latency, etc., except for a few additional wires and registers in each processor. Our method is applicable to both, arrays derived using the standard linear transformation technique and also to those designed in an ad hoc way. Furthermore, since our method is applicable to any systolic array, it is always possible to post-process the derived array based on other design options to derive a fully efficient array without sacrificing the design options used.

Van Dongen [117] is interested in using a class of functions which he also called quasi-linear functions to derive quasi-regular arrays. The class of functions is actually a superset of the class of quasi-linear functions we define. However, his target architectures can be piecewise regular, which is a superset of systolic arrays too. Consequently, his design freedom is much greater than what we have here. It would be interesting to study how to extend our results to quasi-regular arrays.

Recently, there has been some work in the context of merging processors of systolic arrays derived by conventional linear transformation to yield a fully efficient systolic array as reported by Bu and Deprettere [19], and also by Clauss et al. [24]. Both of them adopt the same approach, namely, selecting $\delta - 1$ vectors $\lambda_1, \ldots, \lambda_{\delta-1}$ in the problem domain and merging processors $p_0, p_0 + A\lambda_1, \ldots, p_0 + A\lambda_{\delta-1}$ (where A is the allocation matrix). We call these kinds of approaches "enumeration approach". Our work differs from these approaches in two aspects.

First, due to the selection of vectors $\lambda_1, \ldots, \lambda_{\delta-1}$, it is very possible that $p_0, p_0 +$

$A\lambda_1, \ldots, p_0 + A\lambda_{\delta-1}$ don't form a parallelepiped and may form a cluster of arbitrary shape. Although the resultant array is still regular (because the original array is regular), it is difficult (if not impossible) to come up with an *explicit* allocation function for the final array. Moreover it is not clear how the dense array constraint can be satisfied. In contrast, our approach is constructive and guarantees that an explicit allocation function satisfying the dense array constraint can be found. On the other hand, this method permits the "shape" of the clusters to be arbitrary, and this may allow the user to explore additional clusterings.

Second, the enumeration approaches can only be applied to systolic arrays derived by the conventional linear transformation. In contrast, by studying the activation patterns of the RIA that is implemented by an arbitrary systolic array, our approach can be applied to any (pure) systolic array. Furthermore, the method can be extended to piece-wise systolic arrays as follows. A piece-wise array consists of a constant number of pure systolic subarrays. We can thus apply our technique to each subarray and adjust the connections between boundary processors accordingly. This corresponds to using piece-wise quasi-linear functions, and is especially useful when dealing with ingenuous arrays designed by ad-hoc manner, outside a standard synthesis methodology. Such arrays are most likely to be piece-wise systolic, and our approach can be used to improve the efficiency of such arrays.

Jainundunsing [54], has independently developed a clustering method for two-dimensional arrays (3-d UREs), where he gave a characterization of the periodicity of the behavior of processors within a parallelepiped which is similar to our Lemma 9.10. He thus showed that if appropriate basis vectors are chosen, and if the activation period along them formed a pairwise coprime factorization of δ, there is at most

one active processor within the parallelepiped at any time instant (i.e., there are no timing conflicts). He does not show that there is at least one, and also does not give a method to choose the factorization and the basis. Furthermore, he imposes an additional unnecessary constraint on the timing function. As with the above two approaches, the method is applicable only in the context of synthesis.

Recently, Darte and Delosme have independently reported results similar to ours [31]. They show that it is always possible to cluster a parallelepiped for any factorization of δ. This result is an improvement over Theorem 9.2. However, because our approach is based on the characterization of activation processor-time points, we are able to generalize our result to an arbitrary systolic array. Furthermore, since our approach allows to systematically enumerate all the candidate basis vectors by enumerating all the solutions to a system of diophantine equations, design options can be chosen systematically. It is unclear how to generate all candidate basis vectors in the approach of Darte and Delosme, where the vectors depend on the selection of a unimodular matrix. It would be interesting to see how to combine our approach with theirs.

Our work reported here raises an interesting question regarding the cost (and hence optimality) of systolic arrays derived by linear transformations. Traditionally, the two cost measures that have been used are the computation time and the number of processors. However, by using the results in this chapter, one can always reduce the processor count by a factor of δ. Thus, the "raw" processor count by itself is not an accurate measure. This corresponds to the volume v of a convex polyhedron (the domain of computation, $\mathcal{D}$), under the transformation to space-time, T. Except for two-dimensional recurrences, this is not a linear function and hence the optimal

solution can be obtained only by enumeration. It would be interesting to investigate how such methods [118] can be adapted to use the new cost function which is $v/|T|$.

Besides the possible extensions of our results mentioned above, other open problems related to this study include the sufficient and necessary condition for a quasi-linear function to be timing conflict free and the generalization of the our result on the condition for a quasi-linear function to result in a dense and regular array. Solving these problems gives a complete characterization of quasi-linear allocation functions for systolic design.

CHAPTER X

CONCLUSIONS

In this thesis, we study important issues in mapping a parallel computation to two types of message passing machines, namely, multicomputers and systolic arrays. In the first part, communication overhead issues related to mapping to multicomputers with advanced communication technologies are studied. Empirical studies on the communication overhead related to a mapping are carried out. Message latency prediction formulae are proposed and are validated. A framework for parallel program performance evaluation based on an event-driven simulator is proposed. To reduce communication overhead, we further propose methods to generate routings from the application-specific communication structure for multicomputers with the wormhole routing scheme. The following summarizes the contributions in Part I.

- *The importance of mapping:* we justify that the communication overhead caused due to the topological mismatch between the ideal computation structure for an application and the target architecture can still significantly affect the total completion time of an application for the wormhole routing.

- *Metrics for mapping under wormhole routing:* we empirically characterize effects on the application completion time of path-level contention and dilation, based on simulation results on two widely used benchmarks. Based on our results, we conclude that for wormhole routing, path-level contention is the dominant factor for performance while dilation can not be simply discarded in some cases.

- *Message latency formulae:* we propose and validate message latency formulae which better capture the runtime contention factor as well as dilation. The formulae can be used for simulation overhead reduction as well as for analytical performance studies.

- *Application specific routing algorithms:* we develop application-specific routing algorithms for deadlock-free, low-contention routing. For a general network, we propose a new data structure called GPCDG that captures both channel dependency and channel connection. Two algorithms DFH and $BLOCK$ are developed. For a range of applications and random message distributions, the two algorithms decrease maximum contention by up to 40%.

In the second part, we study three important problems in mapping a class of high-level description algorithms, namely, regular iterative algorithms (RIAs) to systolic arrays. We study issues in schedule (timing) generation, spatial allocation function generation, and the design of efficient systolic arrays. The studies show that a nearly optimal systolic array in all of the above three aspects can be designed for a given RIA. The following summarizes the contributions in Part II.

- *Optimal linear schedule:* the open problem of whether the optimal schedule of a URE can be obtained through a linear or quasi-linear function is answered. The result justifies the use of linear schedules as the timing function in mapping a URE to a systolic array.

- *Linear allocation functions for systolic arrays with limited permissible interconnection:* the number of valid linear allocation functions which result in distinguished arrays is studied and it is found that this number is typically very small.

A framework to design optimal systolic arrays is described based on this result.

- *Efficiency of a systolic array:* we propose a method to find quasi-linear allocation functions to derive systolic arrays which are almost 100% efficient. This result is further extended to any pure systolic array.

While a multicomputer is best for applications with coarse grain parallelism, a systolic array is best for fine grain parallelism. The key difference between these machines lies in their communication methods (the other difference, namely, a systolic array is a synchronous system, is not so fundamental since a wavefront method can be used to achieve systolic computation in an asynchronous system [68]). In a multicomputer, the communication unit for an application is a message with routing information (such as header) while in a systolic array, a stream of data, where each datum is not encapsulated with routing information, can be directly read one by one by the application. Although these two types of machines seem to be disparate, recent development in multicomputer architecture has shown it is possible to support both communication features in a multicomputer [14]. The study of mapping parallel computations onto the two types of architectures helps us to understand the nature of these two types of computations and the techniques proposed above have potential applications in many areas.

Future Work

The work in this thesis opens several interesting new research directions.

- *Simulation-based parallel program performance evaluation:* In Chapter III, we present a scalable and efficient simulation-based evaluation scheme for communication overhead on a multicomputer. By combining our scheme with

Poplawski's synthetic benchmark approach [92], we believe that it is possible to develop a scalable and efficient event driven simulation based parallel program performance evaluation framework. The open problem here is how to estimate the computation cost and incorporate it into an event-driven simulator. Here, we propose an approach which utilizes compile-time data flow analysis information as much as possible and incorporates statistical data on the program runtime behavior either from profiling information or from user input to the simulator.

The first step is to use compile-time data flow analysis to extract information about the computation. For example, we can analyze basic block structures of a program and estimate the computation cost of the basic blocks. This can be accomplished with an instruction-cycle map table to estimate the total cost of a basic block. In practice, the control flow of some dominant computation is only determined by one or two parameters which are the input to the program. For example, in a matrix multiplication, the number of loop iterations in the block submatrix multiplication for each node process is determined by the matrix size and the number of processors used. These two parameters are the input to the whole program. In this case, the whole computation cost can be estimated by the basic block cost (the loop body cost) times the number of loop iterations. Many other benchmarks including FFT, Gaussian elimination, divide and conquer algorithms have such a property.

In the case where the control flow depends on dynamic data in a program, we can use profiling information on the branches (conditions) and replace the conditions on the branches with random variables which reflect statistical information on

the branching conditions. Basically, branch prediction techniques can be used here. Furthermore, user input to these conditions can be used to help determine the branching conditions.

- *Efficient application specific routing and its applications:* In Chapter IV, we present a general framework called the GPCDG to model a multicomputer network with the wormhole routing scheme. More efficient algorithms should be developed to fully explore the properties of a GPCDG and to utilize the properties in developing a good routing. Furthermore, since there are only a handful of regular interconnection networks used in multicomputers (such as meshes, hypercubes and trees), we can explore the regularity of a GPCDG for these specific networks to develop application specific routings.

  Whether one is able to incorporate the application specific routing into a compiler for a parallel program depends on the program language used. For a general C language with message passing capability extension, in many cases, the communication structure for a parallel program can be extracted statically. For some languages, static extraction of the communication structure can be accomplished completely at compile time. These languages include Ada, VHDL and some other data flow languages in digital signal processing. We should study how to apply the application-specific routing technique to these languages.

- *High-level synthesis:* The work presented in the systolic array part can be extended in several directions. First, more general classes of algorithms other than RIA should be studied. For example, one way to extend RIA is to study the problem of synthesizing systolic arrays from structures such as Affine Recurrence Equations where dependency vectors are linear instead of being constant.

In fact, there has been tremendous work done in this direction [97, 96, 21]. We can extend the techniques developed in Part II to solve the problems in synthesis from an ARE.

The mathematical model for the systolic array structure (such as dense array condition and connectivity condition) developed in Part II can be also used in synthesis from a higher-level construct such as an ARE. For example, since a unimodular matrix has many nice mathematical properties, one can expect to utilize such properties to develop better synthesis methods.

Another challenging research area is to study how to apply the techniques developed for systolic array synthesis to a synthesis system for a general digital system (architecture) [53]. We believe that the properties explored on the regular computation structures such as RIAs and AREs can be effectively used in developing synthesis techniques for a general digital system.

- *Parallel compiling techniques:* The techniques developed for systolic array synthesis can be also used in a parallelizing compiler. The result for the optimal scheduling problem indicates that the approach to restructure a DO-loop with constant dependence vectors into an outmost DO-Across loop and several inner DO-All loops can actually capture the maximal parallelism. The quasi-linear allocation functions we proposed for efficient array design can be used in an automatic data decomposition tool for languages such the High Performance Fortran (HPF).

APPENDIX A

BENCHMARK PROGRAMS AND SIMULATION RESULTS

Benchmark Programs

We list the programs of benchmark DAQ and FFT in this section.

Sample DAQ program using PICL-like communication primitives

```
#include "user.h"
#include "picl.h"
#include "daq.h"

#undef EXACT

/**** stereotyping of functions *****/

void ParseArgs();
void node();
int FindSize(int TreeLevel, int MyPhase);
void initialize(int *data, int size);
void divide(int *data, int size, int ith_child,int *childsize);
void compute(int *data, int size);
void combine(int *data, int *childsize);
extern void InitMapping();


/*--------- Global Variables ---------------*/
int *mapping, *imapping,MappingType;
int maxdilation,maxcontention;
int Nx,Ny;

/*--------- Parsing Variables and Flags ----*/
```

```
void ParseArgs(argc, argv)
int  argc;
char *argv[];
{

    /*---- Parsing Flags ------*/
    for (argc--, argv++; *argv && **argv == '-'; argv++, argc--)
    {
switch (*(*argv+1))
{
case 'h':
printf("      -m (reflect (mirror) mapping)\n");
printf ("     -g (growing mapping)\n");
printf ("     -r (random mapping)>\n");
                printf("      -d (default mapping)\n");
exit(1);
break;
case 'm':
MappingType = ReflectMapping;

printf ("using reflect mapping\n");
break;
case 'g':
MappingType = GrowingMapping;
printf ("using growing mapping\n");
break;
case 'r':
MappingType = RandomMapping;
printf ("using random mapping\n");
break;
        case 'd':
                MappingType = DefaultMapping;
                printf("using default (identical) mapping\n");
                break;
        default:
printf ("unknown flag\n");

}
    }

    printf("Nx=");
    scanf("%d", &Nx);
```

```
    printf("\nNy=");
    scanf("%d", &Ny);
    printf("\n");

    /*** PUT INPUT ARGS HERE (E.X. sort data) IF INTENDED **/

}

void node()
{
    int TreeLevel,MyPhase,i;
    int MyNode,Mask,MyPos;
    int data[MaxSize], MySize, ChildSize;
    int myport;
    int MyChild,MyParent;
    int t1,t2;


    TreeLevel=log2(NO_OF_PROCESSORS);

    MyNode=imapping[processor_];


    for(MyPhase=0,Mask=1; MyPhase<=TreeLevel && (MyNode&Mask) !=0;
        Mask *= 2,MyPhase++);

    MySize=FindSize(TreeLevel,MyPhase);

    t1=CURR_TIME;

    open0(&myport);

    if (MyPhase==TreeLevel) {
/* Root */
initialize(data,MySize);
    }
    else {
#ifdef EXACT
recv0(data,MySize,DATA_TYPE,myport);
#else
qrecv0(data,MySize,DATA_TYPE,myport);
#endif
```

```
    }

    MyNode=imapping[processor_];
    MyParent=MyNode+Mask;
    MyPos=log2(Mask)+1;

    for(i=1, Mask /= 2, MyChild=MyNode-Mask; Mask >= 1;
        Mask /= 2,MyChild=MyNode-Mask) {
        divide(data,MySize,i++,&ChildSize);
#ifdef EXACT
        send0(data,ChildSize,DATA_TYPE,mapping[MyChild]);
#else
        qsend0(data,ChildSize,DATA_TYPE,mapping[MyChild]);
#endif
    }

    compute(data,ChildSize);
    for(i=1; i <= MyPhase; i++) {
#ifdef EXACT
        recv0(data,ChildSize,DATA_TYPE+i,myport);
#else
        qrecv0(data,ChildSize,DATA_TYPE+i,myport);
#endif


        combine(data,&ChildSize);
    }

    if (MyPhase != TreeLevel) {
#ifdef EXACT
        send0(data,MySize,DATA_TYPE+MyPos,mapping[MyParent]);
#else
        qsend0(data,MySize,DATA_TYPE+MyPos,mapping[MyParent]);
#endif
    }
    t2=CURR_TIME;

    if (MyPhase==TreeLevel)
        printf("Finishing time for the root=%d Cycles\n", t2-t1);

}
```

```
void usermain(int argc, char** argv)
{
    int i;
    int TreeLevel;

    mysystem_init();

    ParseArgs(argc,argv);
    TreeLevel=log2(Nx*Ny);
    InitMapping(MappingType,TreeLevel);
    load0((FuncPtr)node,-1);
}


/** find data size for the node to process **/
int FindSize(int TreeLevel, int MyPhase)
{
    return BASESIZE;
/*    return BASESIZE*(1 << MyPhase); */
}

/** initialize the data for the root--- do nothing **/
void initialize(int *data, int size)
{
    ;
}


void divide(int *data, int size, int ith_child,int *childsize)
{
    *childsize=size;
}

void compute(int *data, int size)
{
    int i;
    int dummy=1;

    for (i=0;i< size; i++) {
        dummy=dummy*dummy;
    }
}
```

```
void combine(int *data, int *childsize)
{
    int i;
    int dummy=1;

    for (i=0; i<*childsize; i++) {
         dummy=dummy*dummy;
    }
}
```

Sample 2-D FFT program using PICL-like communication primitives

```
#include <math.h>
#include "user.h"
#include "pfft.h"
#include "picl.h"

#undef EXACT

void fft_node();
void reord();
void fft();
void bfly();
void combine();
void complex_sub();
void complex_mult();
int myexp();
int mylog2();
int find_size();
void ParseArgs();
void InitMapping();

/***** variables and types ******/
long *mapping, *imapping,MappingType;

typedef struct {
    long myt;
} Time_Record;

extern Time_Record mytime[NO_OF_PROCESSORS];
```

```
extern Time_Record *that;

/***** program *****/

void fft_node()
{
    int  i,j;
    int  dest;
    complex A[1],B[1],C[1];
    int p,q,p2,p3;
    int Iam;
    int  size;

    int  mask;
    long    StartTime,EndTime,ElapsedTime;
    int   myset,fac;
    int dummy, myport;

/*** every node process has to do this ***/

    open0(&myport);

/*** To simplify, we use the global one for the problem size ***/

    size=find_size();

    /* calculate the cube dimension,size */

    q=NO_OF_PROCESSORS;
    p=mylog2(q);

    /* what is my node number */
    Iam=imapping[processor_];

    /* start the clock */
    if (Iam == 0)
      StartTime = CURR_TIME;


    for (i=Iam,j=0; i<size; i+=q,j++) {
        dummy = dummy+2;
        dummy = dummy-2;
```

```
    }

    p2 = p;
    p3 = mylog2(size);

    /* the first log2(p) phases involve inter PE communication */

    while (p>0) {
        /* complement bits starting from rightmost */
        mask = myexp(2,p2-p);
        if ((Iam & mask) == 0)
            dest = Iam | mask;
        else
            dest = Iam & ~mask;

#ifdef EXACT
        send0(B, sizeof(complex)*size/q, DATA, mapping[dest]);
        recv0(C, sizeof(complex)*size/q, DATA, myport);
#else
        qsend0(B, sizeof(complex)*size/q, DATA, mapping[dest]);
        qrecv0(C, sizeof(complex)*size/q, DATA, myport);
#endif

        myset = Iam % myexp(2,p2-p+1);
        fac = myexp(2,p3-1) * myset;
        if (dest > Iam)
           combine(B,C,B,size/q,fac,size);
        else
           combine(C,B,B,size/q,fac,size);

        p--;p3--;
    }

    /* Now do a sequential fft */
    fft(B,size/q,size,p3);

    ElapsedTime = CURR_TIME - StartTime;
    that->myt=ElapsedTime;

}

int find_size()
```

```
{
    return SIZE*NO_OF_PROCESSORS;
}

void complex_add(A,B,C)
     complex *A,*B,*C;
{
    int dummy;

    dummy=dummy+2;
    dummy=dummy-2;

}

void complex_sub(A,B,C)
     complex *A,*B,*C;

{
    int dummy;

    dummy=dummy+2;
    dummy=dummy-2;

}

void complex_mult(A,B,C)
     complex *A,*B,*C;
{
    complex A1, B1, C1;

    C1.rp = (A1.rp * B1.rp) - (A1.ip * B1.ip);
    C1.ip = (A1.rp * B1.ip) + (A1.ip * B1.rp);
}

/* Exponent that returns Integer */
int myexp(x,y)
     int x,y;
{
    /* raise x to the power y */
    int res = 1;
    while (y>0) {
        res *= x;
```

```
        y--;
    }
    return(res);
}

/* Log to the base 2. Works correctly for powers of 2 only */
int mylog2(x)
     int x;
{
    int y = 0;
    int z = 1;
    if (x < 0)
      return(-1);
    while (z < x) {
        y++;
        z = z << 1;
    }
    return( (z > x) ? y-1 : y);
}

void combine(A,B,C,ct,fac,size)
     complex *A,*B,*C;
     int ct,fac,size;
{
    complex expfac,temp;
    int  i;

    for (i=0; i<(ct/COMM_COMP); i++) {
        expfac.rp = cos((2*PI*fac)/size);
        expfac.ip = sin((2*PI*fac)/size);

        complex_mult(B,&expfac,&temp);
        complex_add(A,&temp,C);

    }
}

void fft(in,size,denom,start)
     complex *in;
     int size,denom,start;
{
    int logsize,i,j,k,r;
```

```
    int Iam,p,q,myset,fac;

    q = start;

    Iam = imapping[processor_];

    p = mylog2(denom);
    logsize = mylog2(size/COMM_COMP);

    for (i=0; i<logsize; i++,start--)  /* log2(N) stages */ {
        myset = Iam % myexp(2,p-start+1);
        fac = myexp(2,start-1) * myset;
        /* span the array */
        for (j=0; j<size; j+=myexp(2,i+1))  {
             /* number of butterflies */
            for (r=fac,k=0; k<myexp(2,i); k++) {
              bfly(in,i,j,k,r,denom);
              r += denom/myexp(2,i+q+1);
            }
        }
    }
}

void bfly(A,i,j,k,r,denom)
     complex *A;
     int i,j,k,r,denom;
{
    int p,q;
    complex expfac,temp1,temp2;
    complex FakeA;

    p = k+j;
    q = p + myexp(2,i);

    expfac.rp = cos(((2*PI*r)/denom));
    expfac.ip = sin(((2*PI*r)/denom));

    complex_mult(&expfac,&FakeA,&temp1);
    complex_add(&FakeA,&temp1,&FakeA);
    complex_sub(&temp2,&temp1,&FakeA);

}
```

```
/******************** Parse Arg **********/
void ParseArgs(argc, argv)
int  argc;
char *argv[];
{

    /*---- Parsing Flags ------*/
switch (*(argv[1]+1))
{
case 'h':
            printf("     -g (gray code mapping) \n");
            printf("     -r (random mapping) \n");
            printf("     -d (default, identical mapping)\n");
            exit(0);
            break;
case 'g':
            MappingType = GrayCodeMapping;
            printf("GRAY CODE:          ");
            break;
case 'r':
            MappingType = RandomMapping;
            printf("RANDOM:             ");
            break;

        case 'd':
            MappingType = DefaultMapping;
            printf("DEFAULT(IDENTICAL): ");
            break;
        default:
            printf ("unknown flag\n");
}

    SIZE=atoi(argv[2]);
    COMM_COMP=atoi(argv[3]);

    printf("SIZE=%-5d COMM_COMP=%d  ", SIZE, COMM_COMP);
}

/*************** usermain ******************/
```

```
void usermain(int argc, char** argv)
{
    int cube_dim;

    mysystem_init();
    define_local(&that, mytime, sizeof(Time_Record));

    ParseArgs(argc,argv);
    cube_dim=mylog2(NO_OF_PROCESSORS);
    InitMapping(MappingType, cube_dim);
    load0((FuncPtr)fft_node,-1);
}
```

Simulation Results

DAQ Simulation Results

The tables in this section show the simulation results for DAQ benchmark. In these tables, the kinds of mapping are represented as: 1. reflecting; 2. growing; 3. random.

Table 13: DAQ Performance on a 1024-Node Wormhole-Routed System

Base size	Mapping	Completion time
2	1	17117
2	2	14381
2	3	22678
128	1	18354
128	2	16288
128	3	23864
8192	1	99017
8192	2	206402
8192	3	104452

Table 14: DAQ Performance on a 256-Node Wormhole-Routed System

Base size	Mapping	Completion time
2	1	7244
2	2	6731
2	3	8670
128	1	8318
128	2	7870
128	3	41294
8192	1	72810
8192	2	119397
8192	3	74213

Table 15: DAQ Performance on a 64-Node Wormhole-Routed System

Base size	Mapping	Completion time
2	1	3956
2	2	3824
2	3	4252
128	1	4712
128	2	4580
128	3	5008
8192	1	53119
8192	2	68282
8192	3	53461

Table 16: DAQ Performance on a 1024-Node Store-Forward Routed System

Base size	Mapping	Completion time
2	1	20226
2	2	16887
2	3	27641
128	1	22935
128	2	19676
128	3	31633
8192	1	206027
8192	2	173669
8192	3	288723

Table 17: DAQ Performance on a 256-Node Store-Forward Routed System

Base size	Mapping	Completion time
2	1	8411
2	2	7580
2	3	162401
128	1	9734
128	2	8650
128	3	12806
8192	1	104657
8192	2	85159
8192	3	141154

Table 18: DAQ Performance on a 64-Node Store-Forward Routed System

Base size	Mapping	Completion time
2	1	4337
2	2	4007
2	3	5052
128	1	5030
128	2	4574
128	3	17526
8192	1	59901
8192	2	50841
8192	3	69935

FFT Simulation Results

The tables in this section show the simulation results for 2-D FFT. In the tables, the size represents the size of the subarray processed in the first phase of the FFT algorithm. The kinds of mapping are represented as 1. gray-code; 2. identical; 3. random. Note, in Table 19, because of the overflow of the cycle counter in the simulator, we are unable to obtain the performance number for the cases where the subarray is of size 4096 bytes.

Table 19: FFT Performance on a 1024-Node Wormhole-Routed System

Size	Mapping	Completion time
8	1	9129
8	2	13527
8	3	35981
64	1	82769
64	2	115995
64	3	250335
512	1	998167
512	2	1282705
512	3	2626767

Table 20: FFT Performance on a 256-Node Wormhole-Routed System

Size	Mapping	Completion time
8	1	9265
8	2	10843
8	3	15485
64	1	70757
64	2	76576
64	3	108162
512	1	876964
512	2	942890
512	3	1148038
4096	1	10472862
4096	2	11000172
4096	3	12599276

Table 21: FFT Performance on a 64-Node Wormhole-Routed System

Size	Mapping	Completion time
8	1	7098
8	2	7953
8	3	8324
64	1	63401
64	2	62452
64	3	67044
512	1	814084
512	2	813135
512	3	888236
4096	1	9951189
4096	2	9950240
4096	3	10436629

BIBLIOGRAPHY

[1] A. Agarwal. Limits on network performance. MIT VLSI Memo, 1992.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Reading,MA, 1974.

[3] F.D. Anger, J. Hwang, and Y. Chow. Scheduling with Sufficient Loosely Coupled Processors. *Journal of Parallel and Distributed Computing*, 9:87–92, 1990.

[4] M. Annaratone, C. Pommerell, and R. Ruhl. Interprocessor communication speed and performance in distributed-memory parallel processors. In *Proceedings of the 16th Annual International Symposium on Computer Architecutre,*, pages 315–324, May 1989.

[5] W.C. Athas and C.L. Seitz. Multicomputers: message-passing concurrent computers. *IEEE Computer*, pages 9–23, August 1988.

[6] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[7] F. Berman. Edge grammars and parallel computation. In *Proceedings of the 1983 Allerton Conference, Urbana, IL*, 1983.

[8] F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, 4(5):439–458, October 1987.

[9] F. Berman and B. Stramm. Prep-p: evolution and overview. Technical Report CS-89-158, Dept. of CS, University of California at San Diego,1989.

[10] B.P. Bianchini and J.P. Shen. Interprocessor traffic scheduling algorithm for multiprocessor networks. *IEEE Trans. Comput.*, C-36(4):396–409, Apr. 1987.

[11] S.H. Bokhari. *Assignment problems in parallel and distributed computing.* Kluwer Academic Publishers, 1987.

[12] S.H. Bokhari. Communication overhead on the intel iPSC-860 hypercube. Technical Report, ICASE, NASA Langley Research Center, May 1990.

[13] S.H. Bokhari. Complete exchange on the iPSC-860. Technical Report, ICASE No. 91-4, NASA Langley Research Center, January 1991.

[14] S. Borkar, R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecutre,*, pages 70–81, May 1990.

[15] A. Borodin and J. Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of Computer and System Sciences*, 30:130–145, 1985.

[16] E.A. Brewer and C.N. Dellarocas. Proteus User Documentation. Laboratory of Computer Science, MIT, 1992.

[17] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, Laboratory of Computer Science, MIT, 1991.

[18] J. Bu and E. Deprettere. Converting sequential iterative algorithms to recurrent equations for automatic design of systolic arrays. In *Proceedings of IEEE International Conference on Accoustics, Speech and Signal Processing*, pages 2025–2028, 1988.

[19] J. Bu, E. Deprettere, and P. Dewilde. A design methodology for fixed-size systolic arrays. In S. Y. Kung and E. Swartzlander, editors, *International Conference on Application Specific Array Processing*, pages 591–602, Princeton, New Jersey, Sept 1990. IEEE Computer Society.

[20] P. Cappello and K. Steiglitz. Unifying VLSI designs with linear transformations of space-time. *Advances in Computing Research*, 2:23–65, 1984.

[21] M. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 3(6):461–491, December 1986.

[22] A. Chien and J. Kim. Planar-adaptive routing: low-cost adaptive networks for multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecutre,*, 1992.

[23] S. Chittor. *Communication Performance of Multicomputers.* PhD thesis, Dept. of Computer Science, Michigan State University, 1991.

[24] P. Clauss, C. Mongenet, and G. R. Perrin. Calculus of space-optimal mappings of systolic algorithms on processor arrays. In S. Y. Kung and E. Swartzlander, editors, *International Conference on Application Specific Array Processing*, pages 4–18, Princeton, New Jersey, Sept 1990. IEEE Computer Society.

[25] E.G.Jr. Coffman (Ed.). *Computer and Job-Shop Scheduling Theory.* John Wiley and Son, New York, 1976.

[26] Intel Scientific Computing. *iPSC2 User Guide.* 1989.

[27] Bailey D.A. and Cuny J.E. Graph grammar based specification of interconnection structures for massively parallel computation. In *Proceedings of the Third International Workshop on Graph Grammars*, pages 73–85, 1987.

[28] W.J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Trans. Comput.*, C-39(6):775–785, June 1990.

[29] W.J. Dally and C.L. Seitz. The torus routing chip. *Distributed Computing*, 1(3):187–196, October 1986.

[30] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks,. *IEEE Trans. Comput.*, C-36(5):547–553, May 1987.

[31] A. Darte and J. Delosme. Partitioning for array processors. Technical Report Report 90-23, Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure De Lyon, France, October 1990.

[32] A. Darte, L. Khachiyan, and Y. Robert. Linear scheduling is nearly optimal. Technical Report LIP Report 91-35, Laboratoire LIP, Ecole Normale Superieure De Lyon, France, November 1991.

[33] G. Duncan. A Comparison of the Performace of Three Message-Routing Strategies for Multicomputers. M.Sc Thesis, Department of Computer Science, University of Edinburgh, 1992.

[34] T.H. Dunigan. Performance of the Intel iPSC/860 and Ncube 6400 hypercubes. *Parallel Computing*, 17:1285–1302, 1991.

[35] S. Duvvuru, R Sundararajan, E. Tick, A.V.S. Sastry, L. Hanson, and X. Zhong. A compile-time memory-reuse scheme for parallel logic programs, lecture notes on computer sciece 637. In *International Workshop on Memory Management.* Springer-Verlag, Sept. 1992.

[36] H. El-Rewini and T.G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[37] R.A. Finkel. Large-grain parallelism - Three case studies. In L. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 21–64, Cambridge, Massachusetts, 1987. The MIT Press.

[38] J.A.B. Fortes and F. Parisi-Presicce. Optimal linear schedule for the parallel execution of algorithms. In *Proc. of 1984 International Conference on Parallel Processing*, 1984.

[39] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fotran D Language Specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[40] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Prentice-Hall, 1988. Volume 1.

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

[42] G.A. Geist, M.T. Heath, Peyton B.W., and P.H. Woeley. PICL: a portable instrumented communication library, C reference manual. Tech. Report ORL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN., July 1990.

[43] D. Gelernter. A dag-based algorithm for prevention of store-and-forward deadlock in packet networks. *IEEE Trans. on Computers*, C-30:709–715, October 1981.

[44] A. Gerasoulis, S. Venugopal, and T. Yang. Clustering Task Graphs for Message Passing Architectures. In *1990 Proceedings of ACM International Congerence on Supercomputing*, pages 447–456, 1990.

[45] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. Techincal Report, Dept. of Computer Science, LCSR-TR-153, September 1990.

[46] C. J. Glass and L. M. Ni. The turn model for adaptive routing. In *Proceedings of the 19th Annual International Symposium on Computer Architecutre,*, 1992.

[47] L. Gravano, G.D. Pifarre, G. Denicolay, and J.L.C. Sanz. Adaptive deadlock-free worm-hole routing in hypercubes. In *Proc. of the Sixth International Parallel Processing Symposium*, pages 512–515, Beverly Hill, California, 1992.

[48] W. G. Griswold, G. A. Harrison, D. Notkin, and L. Snyder. Port ensembles: a communication abstraction for nonshared memory parallel programming. Technical report, University of Washington, 1989.

[49] T. Gross. Communication in iWarp systems. In *Proceedings of Supercomputing'89,*, pages 436–445, November 1989.

[50] M. Hanan and J.M. Kurtzberg. A review of the placement and quadratic assignment problems. *SIAM Review*, 14:324–342, April 1972.

[51] F. Harary. *Graph Theory,*. Addison-Wesley Publishing Company,, 1972.

[52] C.A.R. Hoare. Communicating Sequential Processes. *Communication of ACM*, 21:666–677, 1978.

[53] Synopsys Inc. *Design Compiler Reference Manual.* Synopsys, Inc., Mountain View, California, 1992.

[54] K. Jainundunsing. *Parallel Algorithms for Solving Systems of Linear Equations and Their Mapping on Systolic Arrays.* PhD thesis, Delft University of Technology, Electrical Engineering Department, Delft, the Netherlands, January 1989.

[55] C.R. Jesshope, P.R. Miller, and J.T. Yantchev. High performance communications in processor networks,. In *Proceedings of the 16th Annual International Symposium on Computer Architecutre,*, pages 150–157, June 1989.

[56] L. S. Johnsson. Communication in Network Architectures. In *VLSI and Parallel Processing*, pages 223–389, Chapter 4, 1990.

[57] D.V. Judge and W.G. Rudd. A test case for the parallel programming support environment: parallelizing the analysis of satellite imagery data,. Technical Report, Dept. of CS, Oregon State University,, 1990.

[58] D.D. Kandlur and K.G. Shin. Traffic routing for multi-computer networks with virtual cut-through capability,. In *Preceedings of the 10th International Conference on Distributed Computer Systems,*, pages 398–405, May 1990.

[59] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563–590, July 1967.

[60] S.J. Kim and J.C. Brown. A General approach to mapping of parallel computation upon multiprocessor architectures. In *International Conference on Parallel Processing (ICPP)*, pages 1–8,vol 3, 1988.

[61] K. Koelbel, P. Mehrotra, and J. V. Rosendale. Supporting shared data structures on distributed memory architectures. In *Proceedings of Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, March 1990.

[62] S. Konstantinidou. Adaptive, Minimal Routing in Hypercubes. In *6th MIT Conference on Advanced Research in VLSI*, pages 139–153, 1990.

[63] S. Konstantinidou and L. Snyder. The chaos router: A pratical application of randomization in network routing. In *Proc. of the 1990 ACM Symposium of Parallel Algorithms and Architectures*, pages 79–88, 1990.

[64] S. C. Kothari, H. Oh, and E. Gannett. Optimal designs of linear flow systolic architectures. In *International Conference on Parallel Processing*, St. Charles, Il, 1989. IEEE.

[65] D. Krizanc, S. Rajasekaran, and T. Tsantilas. Optimal routing algorithms for mesh-connected processor arrays. In *VLSI Algorithms and Architectures*, pages 411–422. Springer-Verlag, 1988. Lecture Notes in Computer Science #319.

[66] H. T. Kung. Let's design algorithms for VLSI. In *Proc. Caltech Conference on VLSI*, January 1979.

[67] H. T. Kung and C. E. Leiserson. *Algorithms for VLSI Processor Arrays*, chapter 8.3, pages 271–292. Addison-Wesley, Reading, Ma, 1980.

[68] S. Y. Kung. *VLSI Array Processors.* Prentice Hall, 1988.

[69] S. Y. Kung, S. C. Lo, and P. S. Lewis. Optimal systolic design for the transitive closure and the shortest path problems. *IEEE. Trans. on Computers*, 36(5), May 1987.

[70] S.Y. Lee and J.K. Aggarwal. A mapping strategy for parallel processing. *IEEE Trans. on Computers*, C-36(4):433–442, April 1987.

[71] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufman, 1992.

[72] T. Leighton, F. Makedon, and I. Tollis. A 2n-2 step algorithm for routing in an $n \times n$ array with constant size queues. In *Symposium on Parallel Algorithms and Architectures*, pages 328–335, 1989.

[73] Q. Li. Minimum deadlock-free message routing restriction in binary hypercubes. *Journal of Parallel and Distributed Computing*, 1992.

[74] D.H. Linder and J.C. Harden. An adaptive and fault tolerant wormhole routing strategy for k-ary n-cubes. *IEEE Trans. Comput.*, C-40(1):2–12, January 1991.

[75] L. Lipsett, C. Schaefer, and C. Ussey. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, Boston, 1988.

[76] V. Lo, S. Rajopadhye, J. Telle, and X. Zhong. Mapping divide-and conquer algorithms to parallel architectures. submitted to *IEEE Transactions on Parallel and Distributed Systems*, October 1992.

[77] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. on Computers*, 37(11):1384–1397, 1988.

[78] V.M. Lo. Temporal communication graphs: a new graph theoretic model for mapping and scheduling in distributed memory systems. Sixth Distributed Memory Computing Conference, Portland, Oregon, April 1991.

[79] V.M. Lo, S. Rajopadhye, S. Gupta, D. Kelsen, M.A. Mohamed, and J. Telle. Mapping divide-and-conquer algorithms to parallel architectures. In *Proceedings of International Conference on Parallel Processing*, pages III:128–135, 1990.

[80] V.M. Lo, S. Rajopadhye, S. Gupta, D. Kelsen, M.A. Mohamed, J. Telle, and X. Zhong. OREGAMI: tools for mapping parallel computations to parallel architectures. *International Journal of Parallel Programming*, 20(3), June 1991.

[81] V.M. Lo, S. Rajopadhye, M.A. Mohamed, S. Gupta, B. Nitzberg, J. Telle, and X. Zhong. LaRCS: a Language for Regular Ccommunication Strucutres. *IEEE Trans. on Parallel and Distributed Systems.* To appear.

[82] Inmos Ltd. *occam^{TM} Programming Mannual.* Prentice-Hall, 1984.

[83] F. Makedon and A. Simvonis. On bit-serial packet routing for the mesh and the torus. In *The 3rd Symposium on the Frontier of Massively Parallel Computation*, pages 294–302. IEEE Press, 1990.

[84] U. Manber. *Introduction to Algorithms, A Creative Approach.* Addison-Wesley Publishing Company, 1989.

[85] D. I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proceedings of the IEEE*, 71(1):113–120, January 1983.

[86] D. Nassimi and S. Sahni. An optimal routing algorithm for mesh connected parallel computers. *Journal of the ACM*, 27:6–29, January 1980.

[87] R.M. Nauss. *Parametric Integer Programming.* University of Missouri Press, Columbia, Missouri, 1977.

[88] J.N. Ngai and C.L. Seitz. A Framework for Adaptive Routing in Multicomputer Networks. In *Proc. of the 1989 ACM Symposium of Parallel Algorithms and Architectures*, pages 1–9, 1989.

[89] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput.*, 19(2):322–328, April 1990.

[90] J.L. Peterson and A. Silberschatz. *Operating System Concept.* Addison Wesley Publishing Company, 1990.

[91] C.D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, 1988.

[92] David A. Poplawski. Synthetic models of distributed memory parallel programs. *Journal of Parallel and Distributed Computing*, 1991.

[93] Patrice Quinton. The systematic design of systolic arrays. In *Automata Networks in Computer Science: Theory and Applications*, pages Chpater 9, 229–260. Edited by Soulie, M. and Robert, Y. and Tcheunte, Princeton University Press, 1987. Preliminary versions appear as IRISA Tech Reports 193 and 216, 1983.

[94] S. Rajasekaran and R. Overholt. Constant Queue Routing on a Mesh. In *Proc. Symposium on Theoretical Aspects of Computer Science*, pages 444–455. Springer-Verlag, 1990. Lecture Notes in Computer Science #319.

[95] S. Rajasekaran and M. Raghavachari. Optimal randomized algorithms for multipacket and wormhole routing on the mesh. MS-CIS-91-47, University of Pennsylvania, 1991.

[96] S. V. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, pages 88–105, May 1989.

[97] S. V. Rajopadhye and R. M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14:163–189, June 1990.

[98] S. Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Stanford University, Information Systems Lab., Stanford, Ca, October 1985.

[99] S. Rao and T. Kailath. What is a systolic algorithm. In *Proceedings, Highly Parallel Signal Processing Architectures*, pages 34–48, Los Angeles, Ca, Jan 1986. SPIE.

[100] A.L. Rosenberg. Graph embedding 1988: recent breakthroughs new directions. Technical Report 88-28, University of Massachusetts at Amherst, March 1988.

[101] M. Rosing, R. B. Schnabel, and R.P. Weaver. The dino parallel programming language. Technical Report CU-CS-457-90, Dept. of Computer Science, University of Colorado at Boulder, April 1990.

[102] Gunter Rote. A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion). *Computing*, 34(3):191–219, 1985.

[103] V. Roychowdhury, L. Thiele, S.K. Rao, and T. Kailath. On the localization of algorithms for VLSI processor arrays. In Robert W. Brodersen and Howard S. Moscovitz, editors, *VLSI Signal Processing, III*, pages 459–470, Monterey, Ca, November 1988. IEEE Accoustics, Speech and Signal Processing Society, IEEE Press.

[104] W. Rudd and T.G. Lewis. Architecture of the parallel programming support environment,. In *Proceedings of CompCon'90,*, pages 589–594, San Francisco,CA,, Feb. 1990.

[105] P. Sadayappan, F. Ercal, and J. Ramanujam. Clustering partitioning approaches to mapping parallel programs onto a hypercube. *Parallel Computing*, 13:1–16, 1990.

[106] V. Sarkar. *Partitioning and scheduling parallel programs for multiprocessors.* The MIT Press, 1989.

[107] A. Schrijver. *Theory of Integer and Linear Programming.* John Wiley and Sons, 1988.

[108] C.L. Seitz. The Cosmic Cube. *Communcation of ACM*, 28(1):22–33, January 1985.

[109] W. Shang and J. A. B. Fortes. On the optimality of linear schedules. *Journal of VLSI Signal Processing*, 1:209–220, 1989.

[110] W. Shang and J.A.B. Fortes. Time optimal linear schedules for algorithms with uniform dependencies. *IEEE Trans. on Computers*, 40(6), June 1991.

[111] S.B. Shukla and D.P. Agrawal. Scheduling pipelined communication in distributed memory multiprocessors for real-time applications,. In *Proceedings of the 18th Annual International Symposium on Computer Architecutre,*, pages 222–231, May 1991.

[112] H.S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineeing*, 3(1):85–93, January 1977.

[113] L. Thiele. On the design of piecewise regular processor arrays. In *International Symposium on Circuits and Systems*, pages 2239–2542. IEEE CAS, IEEE Press, 1989.

[114] C.D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Communication of the ACM*, 20:263–271, April 1977.

[115] E. Tick and X. Zhong. A compile-time granularity analysis algorithm and its performance evaluation. *New Generation Computing*, 1993.

[116] L.G. Valiant and G.J. Brebner. Universal schemes for parallel communication. In *Proceedings of 13th Symposium on Theory of Computing*, pages 263–277. ACM, 1971.

[117] V. Van Dongen. Quasi-regular arrays: definition and design methodology. In J. McCanny, J. McWhirter, and E. Swartzlander, editors, *Systolic arrays processors, International Conference on Systolic Arrays*, pages 126–135. Prentice-Hall, 1989.

[118] J. Wong and J. Delosme. Optimization of the processor count for systolic arrays. Technical Report YALEU-DCS-RR-697, Computer Science Dept. Yale University, May 1989.

[119] Y. Yaacoby and P. R. Cappello. Scheduling a system of nonsingular affine recurrence equations onto a processor array. *Journal of VLSI Signal Processing*, 1(2):115–125, 1989.

[120] X. Zhong. Optimal parallel schedules for uniform recurrence equations. Technical Report, CIS-92-12, Computer Science Dept., University of Oregon, April 1992.

[121] X. Zhong and V. M. Lo. Application-specific deadlock free wormhole routing on multicomputers. In *Parallel Architectures and Languages, Europe (PARLE'92), Lecture Notes in Computer Science*, pages 264–277, Paris,France, June 1992. Springer-Verlag.

[122] X. Zhong and V. M. Lo. An efficient heuristic for application-specific routing on mesh connected multiprocessors. In *International Conference on Parallel Processing (ICPP)*, Chicago, IL, August 1992.

[123] X. Zhong, V. M. Lo, and S.V. Rajopadhye. Optimal implementation of divide-and-conquer algorithms on binary de bruijn networks. 4th Symposium on the Frontier of Massively Parallel Computations, March 1992.

[124] X. Zhong and S. V. Rajopadhye. Deriving fully efficient systolic arrays by quasi-linear allocation functions. In *Parallel Architectures and Languages, Europe, 1991, Lecture Notes in Computer Science 505, Springer-Verlag*, pages 219–235, June 1991.

[125] X. Zhong and S. V. Rajopadhye. Synthesizing efficient systolic arrays. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, Toronto, Canada, May 1991. IEEE.

[126] X. Zhong and S.V. Rajopadhye. Quasi-linear allocation functions for efficient array design. *Journal of VLSI Signal Processing*, 4:97–110, 1992.

[127] X. Zhong, S.V. Rajopadhye, and V.M. Lo. Parallel implementation of divide-and-conquer algorithms on binary de bruijn networks. In *Proc. of the Sixth International Parallel Processing Symposium*, pages 103–107, Beverly Hill, California, 1992.

[128] X. Zhong, S.V. Rajopadhye, and I. Wong. Systematic generation of linear allocation functions in systolic array design. *Journal of VLSI Signal Processing*, 4:279–293, 1992.

[129] X. Zhong, E. Tick, S. Duvvuru, A.V.S. Sastry, and R. Sundararajan. Towards an efficient compile-time granularity analysis algorithm. In *International Conference on Fifth Generation Computer Systems*, pages 809–816. ICOT,Tokyo, June 1992.

[130] X. Zhong, I. Wong, and S. V. Rajopadhye. Bounds on the number of linear allocation functions. In *VLSI Signal Processing IV*, pages 85–94, San Diego, CA, November 1990. IEEE ASSP Society.