Dynamic and Compiled Communication in Optical Time–Division–Multiplexed Point–to–Point Networks

by

Xin Yuan

B.S., Shanghai Jiaotong University, 1989

M.S., Shanghai Jiaotong University, 1992

M.S., University of Pittsburgh, 1995

Submitted to the Graduate Faculty of
Arts and Sciences in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

University of Pittsburgh

1998

UNIVERSITY OF PITTSBURGH

FACULTY OF ARTS AND SCIENCES

This dissertation was presented

by

Xin Yuan

It was defended on

August, 1998

and approved by

Prof. Rajiv Gupta (Co–Chair)

Prof. Rami Melhem (Co–Chair)

Prof. Henry Chuang

Prof. Thomas Gross

Committee Chairperson(s)

Dynamic and Compiled Communication in Optical Time–Division–Multiplexed Point–to–Point Networks

Xin Yuan, Ph.D.

University of Pittsburgh, 1998

Optical interconnection networks are promising networks for future supercomputers due to their large bandwidths. However, the speed mismatch between the fast optical data transmission and the relatively slow electronic control components poses challenges for designing an optical network whose large bandwidth can be utilized by end users. The Time–Division–Multiplexing (TDM) technique alleviates this mismatch problem by sacrificing part of the large optical bandwidth for efficient network control. This thesis studies efficient control mechanisms for optical TDM point–to–point networks. Specifically, three communication schemes are considered, dynamic single–hop communication, dynamic multi–hop communication and compiled communication.

Dynamic single–hop communication uses a path reservation protocol to establish all–optical paths for connection requests that arrive at the network dynamically. An efficient path reservation protocol is essential for this scheme to achieve high performance. In this thesis, a number of efficient distributed path reservation protocols are designed and the impact of system parameters on these protocols is studied.

Dynamic multi–hop communication allows intermediate hops to route messages toward their destinations. In optical TDM networks, efficient dynamic multi–hop communication can be achieved by routing messages through a logical topology that is more efficient than the physical topology. This thesis studies efficient schemes to realize logical topologies on top of physical torus topologies, presents an analytical model for analyzing the maximum throughput and the average packet delay for multi–hop networks, and evaluates the performance of the optical communication on the logical topologies.

Compiled communication eliminates the runtime communication overheads of the dynamic communications by managing network resources at compile time. This thesis considers issues for applying the compiled communication technique to optical TDM networks, including communication analysis, connection scheduling and communication phase analysis. A compiler, called the E–SUIF compiler, is implemented to support compiled communication on optical TDM networks.

Each communication scheme has its advantages and limitations and is more efficient for some types of communication patterns. This thesis compares the performance of the three communication schemes using a number of benchmarks and real application programs and identifies the situations where each communication scheme out–performs the other schemes.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Fiber–optic technology has advanced significantly over the past few years, so have the development of tunable lasers, filters, high–speed transmitter and receiver circuits, optical amplifiers and photonic switching devices [40, 76]. With the maturing of optical technology, transmission cost, and in particular the cost of high speed data links, has dropped tremendously. The electronic processing capability of computers cannot match the potentially very high speed of optical data transmission. The communication bottleneck has shifted from the transmission medium to the processing needed to control that medium.

In optical interconnection networks, each physical link can offer very high bandwidth. In order to fully utilize the available bandwidth, an optical link can be shared through *time–division multiplexing* (TDM) [23, 53, 61] and/or *wavelength–division multiplexing* (WDM) [9, 19, 22, 72]. In TDM, optical links are multiplexed by assigning different virtual communication channels to different *time slots*, while in WDM, optical links are multiplexed by assigning different virtual communication channels to different *wavelengths.* By using TDM, WDM or TWDM (a combination of TDM and WDM), each link can support multiple channels.

Point–to–point networks, such as meshes, tori, rings and hypercubes, are used in commercial supercomputers. By exploiting space diversity and traffic locality, they offer larger aggregate throughput and better scalability than shared media networks such as buses. Optical point–to–point networks can be implemented by replacing electronic links with optical links and operating in a packet switching fashion just like electronic networks. The performance of such networks is limited by the speed of electronics since buffering and address decoding are performed in the electronic domain. Thus, these networks cannot efficiently utilize the potentially high bandwidth that optics can provide. New generation optical point–to–point networks exploit the *channel–routing* capability in optical switches. In TDM networks, time–slot routing[61] is used, while in WDM networks, wavelength–routing[20] is used. The channel routing in an optical switch routes messages from a channel of an input port to a channel of an output port without converting the messages into the electronic domain. The switch states, however, are usually controlled by electronic signals.

Using channel routing, two approaches can be used to establish connections in multiplexed optical networks, namely *link multiplexing* (LM) and *path multiplexing* (PM). These connections are called *lightpaths* since the light signal travels through the connections that may span a number of optical links and switches without being converted into the electronic domain. In LM, a connection which spans more than one communication link is established by using possibly different channels on different links. In PM, a connection which spans more than one communication link uses the same channel on all the links. In other words, PM uses the same time-slot or the same wavelength on all links of a connection, while LM can use different time-slots or different wavelengths, thus requiring time-slot interchange or wavelength conversion capabilities at each intermediate optical switch. Since the technology to support PM is more mature than the one to support LM, this thesis focuses on PM.

Since the electronic processing speed is relatively slow compared to the optical data transmission speed, optical point–to–point networks should ideally employ *all–optical* communication in data transmission. In all–optical communication, no electronic processing and no electronic/optical (E/O) or optical/electronic (O/E) conversions are performed at intermediate nodes. Once converted into the optical domain, the signal remains there until it reaches the destination. All–optical communication eliminates the electronic processing bottleneck at intermediate nodes during data transmission and thus, exploits the large bandwidth of optical links. This thesis considers all–optical networks where a lightpath is established before a communication starts and the data transmission is carried out in a pure circuit–switching fashion. This type of communication is referred to as the *dynamic single–hop communication*. In such networks, electronic processing occurs only in the path reservation process and hence, using an efficient path reservation protocol is crucial to obtain high performance. In this work, a number of path reservation algorithms that dynamically establish lightpaths are designed and the impact of system parameters on the algorithms is studied. These algorithms use a separate control network to exchange control messages and allow all–optical communication in the optical data network.

Although dynamic single–hop networks achieve all–optical communication in data transmission, the path reservation algorithms require extra hardware support to exchange control messages and result in large startup overhead, especially for small messages. An alternative is to use *dynamic multi–hop communication*. In multi–hop networks, intermediate nodes are responsible for routing packets such that a packet sent from a sender will eventually reach its destination, possibly after being routed through a number of intermediate nodes. Clearly, multi–hop networks require E/O and O/E conversions at intermediate nodes. Thus, it is important to reduce the number of hops that a packet visits. This reduction may be achieved in an optical TDM network by combining the channel–routing technique and the packet switching technique. Specifically, packets may be routed through a logical topology which has a small diameter as opposed to the physical topology which may have a large diameter. The major issue in the multi–hop communication is to design appropriate logical topologies. This thesis considers efficient schemes for realizing logical topologies on top of physical mesh and torus networks using path multiplexing. Realizing logical topologies on optical networks is different from traditional embedding techniques in that both routing and channel assignment options must be considered. An analytical model that models the maximum throughput and average package latency of multi–hop networks is developed and is used to evaluate the performance of logical topologies and identify the advantages of each logical topology.

While dynamic (single–hop or multi–hop) communications handle arbitrary communication patterns, their performance can be limited by the electronic processing which occurs during path reservation in single–hop communication and during packet routing in multi–hop communication. *Compiled communication* overcomes this limitation for communication patterns that are known at compile time. In compiled communication, the compiler analyzes a program and determines its communication requirement. The compiler then uses the knowledge of the underlying architecture, together with the knowledge of the communication requirement, to manage network resources statically. As a result, runtime communication overheads, such as path reservation and buffer allocation overheads, are reduced or eliminated, and the communication performance is improved. However, due to the limited network resources, the underlying network cannot support arbitrary communication patterns. Compiled communication requires for the compiler to analyze a program and partition it into phases such that each phase has a fixed, pre-determined communication pattern that the underlying network can support. The compiler inserts codes for performing network reconfigurations at phase boundaries to support all connections in the next phase. At runtime, a lightpath is available for each communication without path reservation. Therefore, compiled communication accomplishes all–optical communication without incurring extra hardware support and large start–up overheads. This thesis studies the application of compiled communication to optical interconnection networks. Specifically, it considers the communication analysis techniques needed to analyze the communication requirement of a program. These analysis techniques are general in that they can be applied to other communication optimizations and can be used for compiled communication in electronic networks. This thesis also develops a number of connection scheduling schemes which realize a given communication pattern with a minimal multiplexing degree. Note that in optical TDM networks, communication time is proportional to the multiplexing degree. Finally, a communication phase analysis algorithm is developed to partition a program into phases so that each phase contains connections that can be supported by the underlying network. All the algorithms are implemented in a compiler which is based on the Stanford SUIF compiler[3]. This thesis evaluates the performance of the algorithms in terms of both analysis cost and runtime efficiency.

	Single–hop	Multi–hop	Compiled
All–optical comm.	Yes	No	Yes
Startup overhead	Yes	No	No
Extra hardware	Yes	No	No
Arbitrary comm.	Yes	Yes	No

Table 1.1: General characteristics of the three schemes

Table 1.1 summarizes the general characteristics of the three schemes. In optical interconnection networks, the central problem to be addressed is the reduction of the amount of electronic processing needed for controlling the communication. In dynamic single–hop networks, this problem is addressed by having efficient path reservation algorithms. In multi–hop networks, this problem is tackled by designing efficient logical topologies to route messages. Compiled communication totally eliminates the electronic processing in communications. However, it only applies to the communication patterns that are known at compile time. While communications in optical TDM point–to–point networks can be carried out by any of the three communication schemes, it is necessary to understand the strengths and the limitations of each communication scheme in order to make appropriate choices when designing an optical interconnection network. In addition to considering the options within each communication scheme, this thesis compares the performance of the three communication schemes using a number of benchmarks and real applications and identifies the situations in which each communication scheme has advantage over other schemes.

The remainder of the thesis is organized as follows. Chapter 2 begins by describing background and thesis assumptions. This chapter presents an overview of optical interconnection networks, discusses the TDM technique and introduces the *path multiplexing* (PM) and *link multiplexing* (LM) techniques for establishing connections. This chapter also surveys the research related to the three communication schemes. Finally, this background chapter surveys the traditional compilation techniques for distributed memory machines and communication optimizations and discusses the difference between the traditional communication optimizations and the compiled communication technique.

Chapter 3 discusses the techniques used in dynamic single–hop communication. Two types of distributed path reservation protocols, the *forward reservation protocols* and the *backward path reservation protocols*, are described. This chapter also describes a network simulator for dynamic single–hop communication, evaluates the performance of the two types of protocols and studies the impact of system parameters on these protocols.

Chapter 4 discusses dynamic multi–hop communication. This chapter presents efficient schemes for realizing logical topologies on top of the physical torus networks, describes an analytical model that models the maximum throughput and the average packet delay for the logical topologies and verifies the model with simulation. In addition, this chapter also describes the simulator for dynamic multi–hop communication, evaluates the multi–hop communication with the logical topologies and identifies the advantages and the limitations of each logical topology.

Chapter 5 considers compiled communication. This chapter describes the communication analyzer and discusses the communication descriptor used in the analyzer, the data flow analysis algorithms for communication optimizations, and the actual communication optimization performed in the analyzer. The chapter also describes off-line connection scheduling algorithms and a communication phase analysis algorithm. Using these algorithms, the compiler analyzes the communication requirement of a program, partitions the program into phases such that each phase contains connections that can be supported by the underlying network, and schedules the connections within each phase. The chapter also presents the evaluation of the compiler algorithms and studies their runtime efficiency.

Chapter 6 compares the communication performance of the three communication schemes. Three sets of application (benchmark) programs, including hand–coded parallel programs, HPF benchmark programs and sequential programs from SPEC95, are used to evaluate the communication performance of the communication schemes. Different sets of programs exhibit different communication characteristic. For example, the hand–coded programs are highly optimized for parallel execution, while the programs from SPEC95 are not optimized for parallel execution. This chapter compares the communication performance of the three communication schemes and identifies the advantages of each scheme.

Finally, Chpater 7 summarizes the dissertation and suggests some directions for future research.

Chapter 2

Background and related work

2.1 Optical TDM networks

An optical point–to–point network consists of switches with a fixed number of input and output ports. One input port and one output port are used to connect the switch to a local processing element and all remaining input and output ports of a switch are used for connections to other switches. An example of such networks is the 4×4 torus shown in Figure 2.1. In these networks, each link in the network is time–multiplexed to support multiple virtual channels.

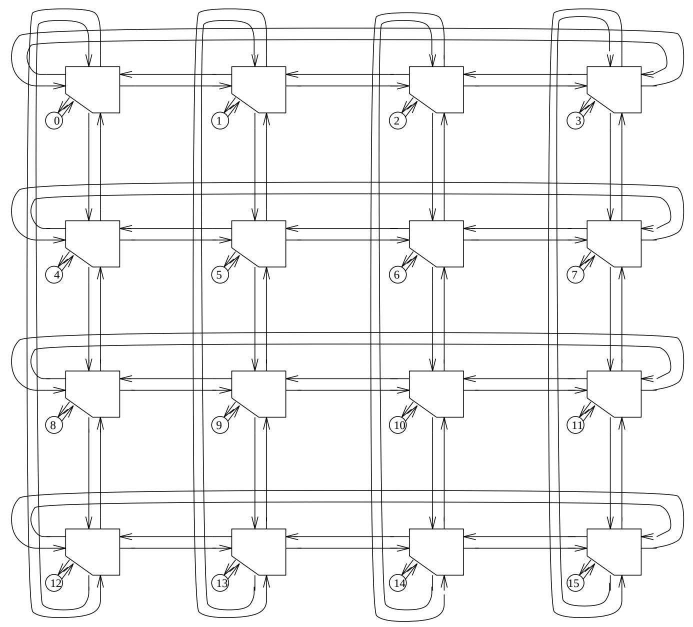


Figure 2.1: A torus connected network

Two approaches can be used to establish connections in multiplexed networks, namely *link multiplexing* (LM) and *path multiplexing* (PM) [62]. PM uses the same channel on all the links along the path to form a connection. On the other hand, LM may use different channels on different links along the path, thus requiring time-slot interchange in TDM networks at each intermediate node. Fig. 2.2 shows the PM and LM connections at a 2×2 switch where each link supports two channels. LM is similar to the multiplexing technique in electronic networks where a data packet can change channels when it passes a switch. Using LM for communication has many advantages over using PM. For example, the path reservation for a LM connection is simpler than that for a PM connection, and LM results in better channel utilization. However, optical devices for LM are still in the research stage and are very expensive using current technology. Hence, this thesis is concerned only with path multiplexing because the enabling technology is more mature.

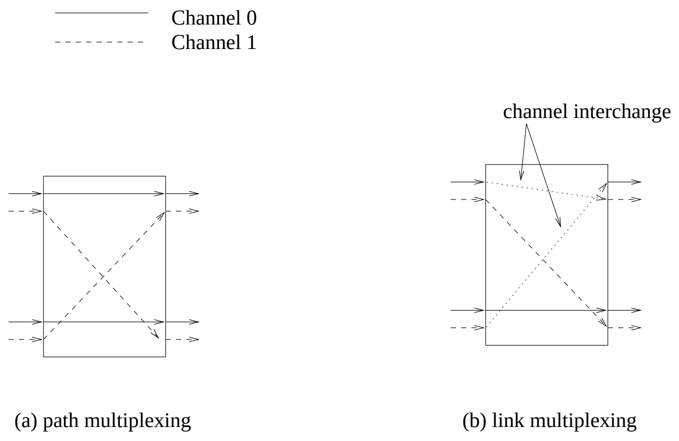


Figure 2.2: Path multiplexing and link multiplexing

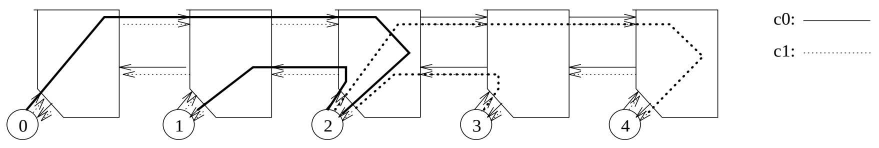


Figure 2.3: Path multiplexing in a linear array

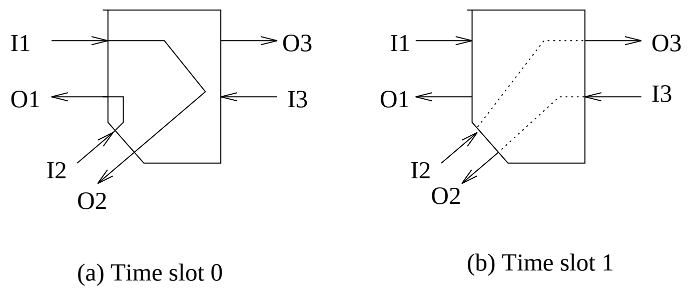


Figure 2.4: Changing the state of a switch in TDM

In order to time–multiplex a network with path multiplexing, a *time slot* is defined to be a fixed period of time and the time domain is divided into a repeated sequence of

d time slots, where d is the *multiplexing degree.* Different virtual channels on each link occupy different time slots. Figures 2.3 and 2.4 illustrate path multiplexing on a linear array. In these two figures, two virtual channels are supported on each link by dividing the time domain into two time slots, and using alternating time slots for the two channels $c0$ and $c1$. Let us use (u, v) to denote a connection from node u to node v. Figure 2.3 shows four established connections over the two channels, namely connections $(0, 2)$ and $(2, 1)$ that are established using channel $c0$, and connections $(2, 4)$ and $(3, 2)$ that are established using channel $c1$. The switches, called *Time–Multiplexed Switches* (TMS), are globally synchronized at time slot boundaries, and each switch is set to alternate between the two states that are needed to realize the established connections. For example, Figure 2.4 shows the two states that the 3×3 switch attached to PE 2 must realize for the establishment of the connections shown in Figure 2.3. Note that each switch can be an electro-optical switch (Ti:LiNbO$_3$ switch, for example [36]) which connects optical inputs to optical outputs without E/O and O/E conversions. The state of a switch is controlled by setting electronic bits in a *switch state register.*

The duration of a time slot may be equal to the duration over which several hundred bits may be transmitted. For synchronization purposes, a guard band at each end of a time slot must be used to allow for changing the state of switches (shifting a shift register) and to accommodate possible drifting or jitter. For example, if the duration of a time slot is $276ns$, which includes a guard band of $10ns$ at each end, then $256ns$ can be used to transmit data. If the transmission rate is $1Gb/s$, then a packet of 256 bits can be transmitted during each time slot. Note that the optical transmission rate is not affected by the relatively slow speed of changing the state of switches ($10ns$) since that change is performed only every $276ns$.

Communications in TDM networks can either be *single–hop* or *multi–hop.* In single–hop communication, circuit–switching style communications are carried out. A path for a communication must be established before the communication starts. In general, any $N \times N$ network, other than a completely connected network, has a limited connectivity in the sense that only subsets, $C = \{(x, y)|0 \leq x, y < N\}$, of the possible N^2 connections can be established simultaneously without conflict. For single–hop communication the network must be able to establish any possible connection in one hop, without intermediate relaying or routing. Hence, the network must be able to change the connections it supports at different times. This thesis considers switching networks in which the set of connections that may be established simultaneously (that is, the state of the network) is selected by changing the contents of hardware registers. The single–hop communication can be achieved in two ways. First, a path reservation algorithm can be used to dynamically establish and tear down all–optical connections for arbitrary communications. Second, compiled communication uses the compiler to analyze the communication requirement of a program and insert code to establish all–optical connections (at phase boundaries) before communications start. Unlike the case in a single–hop system where connections are dynamically established and torn down, connections in a multi–hop system are fixed and a message may travel through a number of lightpaths to reach its destination. Dynamic single–hop communication, dynamic multi–hop communication and compiled communication will be discussed in some details next.

2.2 Dynamic single–hop communication with PM

To establish a connection in an optical TDM network, a physical path, PP, from the source to the destination is first chosen. Then, a virtual path, VP, consisting of a virtual channel in each link in PP is selected and the connection is established. The selection of PP has been studied extensively and is well understood [50]. It can be classified into *deterministic* routing, where PP can be determined from the source node and the destination node (e.g., X–Y routing on a mesh), or *adaptive* routing, where PP is selected from a set of possible paths. Once PP is selected, a time slot is used in all the links along PP.

The control in optical TDM networks is responsible for the establishment of a virtual path for each connection request. Due to the similarity of TDM and WDM networks, many techniques for virtual channel assignment in one of these two types of networks can also apply to the other type. Network control for multiplexed optical networks can be classified into two categories, centralized control and distributed control. Centralized control assumes a central controller which maintains the state of the whole network and schedules all communication requests. Many time slot assignment and wavelength assignment algorithms have been proposed for centralized control. In [61] a number of time slot assignment algorithms are proposed for TDM multi-stage interconnection networks. In [19] wavelength assignment for wide area networks is studied. A time wavelength assignment algorithm for WDM star networks is proposed in [23]. Theoretical study for optimal routing and wavelength assignment for arbitrary networks is presented in [64].

Distributed control does not assume a central controller and thus is more practical for large networks. Little work has been done on distributed control for optical multiplexed networks. In [61] a distributed path reservation scheme for optical Multistage Interconnection Networks (MIN) is proposed. Distributed path reservation methods for both path multiplexing and link multiplexing are presented in [63]. This thesis proposes distributed path reservation algorithms that are more efficient than the previous algorithms, investigates variations in channel reservation methods and studies the impact of the system parameters on the protocols.

2.3 Dynamic multi–hop communication with PM

By using path multiplexing, efficient logical topologies can be established on top of the physical topology. The connections in the logical topologies are lightpaths that may span a number of links. In such systems, the switching architecture consists of an optical component and an electronic component. The optical component is an all–optical switch, which can switch the optical signal from some input channels to output channels in the optical domain (i.e., without E/O and O/E conversions), and which can locally terminate some other lightpaths by directing them to the node's electronic component. The electronic component is an electronic packet router which serves as a store–and–forward electronics overlaid on top of the optical virtual topology. Figure 2.5 provides a schematic diagram of the architecture of the nodal switch in a physical torus topology.

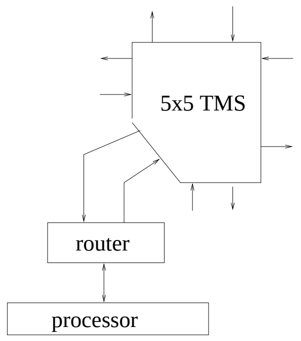


Figure 2.5: A nodal switching architecture

Since the electronic processing is slow compared to the optical data transmission, it is desirable to reduce the number of intermediate hops in a multi–hop network. This can be achieved by having a logical topology whose connectivity is high. However, realizing a logical topology with a large number of connections requires a large multiplexing degree. In a TDM system, large multiplexing degree results in a large time to transmit a packet through a lightpath because every light path is established only for a fraction of the time. Hence, there exists a performance trade–off in the logical topology design between a logical topology with large multiplexing degree and high connectivity and a logical topology with small multiplexing degree and low connectivity. As will be shown in this dissertation, both topologies have advantages for certain types of communication patterns and system settings.

Multi–hop networks have been extensively studied in the area of WDM wide area networks. The works in [6, 20, 47, 57, 77] consider the realization of logical topologies on optical multiplexed networks. These works consider wide area networks and focus on designing efficient logical topologies on top of irregular networks. Since finding an optimal logical topology on irregular networks is an NP–hard problem, heuristics and simulated annealing algorithms are used to find suboptimal schemes. This dissertation considers regular networks in multiprocessor environments and derives optimal connection scheduling schemes for realizing hypercube communications. Besides logical topology design, connection scheduling algorithms can also be used to realize logical topologies. In [63] message scheduling for permutation communication patterns in mesh–like networks is considered. In [61] optimal schemes for realizing all–to–all patterns in multi-stage networks are presented. In [33] message scheduling for all–to–all communication in mesh–like topologies is described.

The performance of multi–hop networks has also been previously studied. However, most previous performance studies for optical multi–hop networks assume a broadcast based underlying WDM network, such as an optical star network [45, 67], where the major concerns are the number of transceivers in each node and the tuning speed of the transceivers. This thesis studies the logical topologies on top of a physical torus topology in a TDM network, where the major focus is the trade–off between the multiplexing degree and the connectivity of a topology.

2.4 Compiled communication

Compiled communication has recently drawn the attention of several researchers [13, 34]. Compiled communication has been used in combination with message passing in the iWarp system [25, 26, 35], where it is used for specific subsets of static patterns. All other communications are handled using message passing. The prototype system described in [13] eliminates the cost of supporting multiple communication models. It relies exclusively upon compiled communication. However, the performance of this system is severely limited due to frequent dynamic reconfigurations of the network. Compiled communication is more beneficial in optical multiplexed networks. Specifically, it reduces the control overhead, which is one of the major factors that limit the communication performance in optical networks. Moreover, multiplexing, which is natural in optical interconnection networks, enables a network to support simultaneously more connections than a non–multiplexed network, which reduces the reconfiguration overhead in compiled communication.

The communication patterns in an application program can be broadly classified into two categories: *static patterns* that can be recognized by the compiler and *dynamic* *patterns* that are only known at run-time. For a static pattern, compiled communication computes a minimal set of network configurations that satisfies the connection requirement of the pattern and thus, handles static patterns with high efficiency. Recent studies [48] have shown that about 95% of the communication patterns in scientific programs are static patterns. Thus, using the compiled communication technique to improve the communication performance for the static patterns is likely to improve the overall communication performance. Some advantages of using compiled communication for handling static patterns are as follows.

- Compiled communication totally eliminates the path reservation and the large startup overhead associated with the path reservation.
- The connection scheduling algorithm is executed off-line by the compiler. Therefore, complex strategies can be employed to improve network utilization.
- No routing decisions are made at runtime which means that the packet header can be shortened causing the network bandwidth to be utilized more effectively.
- Optical networks efficiently support multiplexing which reduces the chance of network reconfigurations due to the lack of network capacity.
- Compiled communication adapts to the communication requirement in each phase. For example, it can use different multiplexing degrees for different phases in a program. In contrast, dynamic communications always use the same configuration to handle all communications in a program which may not be optimal.

In order to apply compiled communication to a large scale multiprocessor system, three main problems must be addressed:

Communication Pattern Recognition: This problem has been considered by many researchers since information on communication patterns has been previously used to perform communication optimizations [11, 28, 34, 51]. The stencil compiler [11] for CM-2 recognizes *stencil* communication patterns. Chen and Li [51] incorporated a pattern extraction mechanism in a compiler to support the use of collective communication primitives. Techniques for recognizing a broad set of communication patterns were also proposed in [28]. However, most of these methods determine a specific subset of static communication patterns, such as the broadcast pattern and the nearest neighbor pattern, which is not sufficient for compiled communication. Since the communication performance of compiled communication relies heavily on the precision of the communication analysis, it is desirable to perform more precise analysis that can recognize arbitrary communication patterns. Furthermore, compiled communication requires the partitioning of a program into phases, such that each phase contains communications that can be supported by the underlying network, and the scheduling of connections within each phase. These are new problems that must be addressed.

Compiling Static Patterns: Once the compiler determines a communication pattern within each phase, which is called a *static pattern*, the compiler must be able to schedule the communication pattern on the multiplexed network. In TDM networks, communication performance is proportional to the multiplexing degree. Given a communication pattern, the smaller the multiplexing degree, the less time the communication lasts. Thus, connection scheduling algorithms that schedule all connection requests in a phase with a minimal multiplexing degree must be designed to handle the static patterns. It has been shown that optimal message scheduling for arbitrary topologies is NP-complete [19]. Hence, heuristic algorithms that provide good performance need to be developed.

Handling Dynamic Patterns: A number of techniques can be used to handle dynamic communication patterns. One approach is to setup all-to-all connections among all nodes in the system. This way each node has a time slot to communicate with every other node. However, establishing paths for the all-to-all communication can be prohibitively expensive for large systems. An alternative is to perform dynamic single–hop or multi-hop communications. The dynamic communications are not as efficient as compiled communication. However, since this method is not used frequently, its effect on the overall performance is limited.

2.5 Programming and machine model

Compiled communication requires the compiler to extract communication patterns from application programs. The method to extract communication patterns in a program depends on both programming model and machine model. The programming model includes the ones using explicit communication primitives and the ones that require implicit communication through remote memory references. There are two different machine architectures, the shared memory machine and the distributed memory machine. Communication requirements for these two machine models are different for a program. In shared memory machines with hardware cache coherence, communications result from cache coherence traffic, while in distributed memory machines, communications result from data movements between processors.

Explicit communication: Most of the current commercial distributed memory supercomputers support the explicit communication programming model. In such programs, programmers explicitly use communication primitives to perform the communication required in a program. The communication primitives can be high level library routines, such as PVM [60] or MPI [54], or low level communication primitives such as the shared memory operations in the CRAY T3D [59] and the CRAY T3E. Communication patterns in a program with explicit communication primitives can be obtained from the analysis of the communication primitives in the program.

Implicit communication: Managing explicit communication is tedious and error-prone. This has motivated considerable research towards developing compilers that relieve programmers from the burden of generating communication [2, 5, 31, 37, 65, 88]. Such compilers take sequential or shared memory parallel programs and generate Single Program Multiple Data (SPMD) programs with explicit message passing. This type of programs will be referred to as *shared memory programs*. Shared memory programs can be compiled for execution on both distributed memory machines and shared memory machines. In the case when a program is to be run on a distributed memory machine, the communication requirements of the program can be obtained from memory references. If a program is to be run on a shared memory machine, the communication requirements depend on the cache behavior. However, a superset of the communication patterns may be obtained by examining the memory references in the program. This work will consider data parallel shared memory programs compiled for execution on distributed memory machines. Compilers that exploit task parallelism [27, 71] are not considered. However, similar techniques may also apply to task parallel programs.

2.6 Compilation for distributed memory machines

While communication requirements of a shared memory program can be obtained by analyzing the remote memory references in the program, the actual communication patterns in the program depend on the compilation techniques used. To obtain realistic communication patterns, compilation techniques for compiling shared memory programs for distributed memory machines must be considered. The most important issues to be addressed when compiling for distributed memory machines are data partitioning, code generation and communication optimization. This section surveys previous work on these issues.

Data partitioning decides the distribution of array elements to processors. There are two approaches for handling the data partitioning problem. The first approach is to add user directives to programming languages and let the users specify the data distribution. This approach is used in Fortran D [37], Vienna Fortran [15] and High Performance Fortran (HPF) [39] among others. It uses human knowledge of application programs and simplifies the compiler design. However, using this approach requires programmers to work at a low level abstraction (understanding the detail of memory layout). Since the best placement decision will vary between different architectures, with explicit user placement, the programmer must reconsider the data placement for each new architecture. Hence, many algorithms have been developed to perform automatic data distribution. An algorithm has been designed for the CM Fortran compiler that attempts to minimize and identify alignment communications in data parallel Fortran programs [42]. Similar algorithms have been proposed in [16, 29, 34, 52]. Data partitioning directly affects the communication requirements in a program running on a distributed memory machine. Once data partitioning is decided, the minimum requirement of data movements in a program, which results in communications, is fixed.

Code generation generates the communication code to ensure the correctness of a program. The *Owner computes* rule is generally used for distributing the computation onto processors. Under owner computes rule, the owner of the array element on the left hand side of an assignment statement executes the statement. Thus, the owner of an array element on the right hand side of the assignment statement must send the element to the owner of the left hand side, which results in communication. Without considering efficiency, a simple scheme can be used to generate the correct SPMD code by inserting guarded communication primitives [65]. However, the communication and synchronization overhead of this scheme can be so large that there may be no benefit for running the program on a multiprocessor system. Several researchers have proposed techniques for generating efficient code for array statements, given *block*, *cyclic* and *block–cyclic* distributions. In [43, 44] compile time analysis of array statement with block and cyclic distribution is presented. In [17] Chatterjee et al. present a framework for compiling array assignment statements in terms of constructing a finite state machine. This method handles block, cyclic and block–cyclic distributions. Method in [69] improves Chatterjee's method in terms of buffer space and communication code generation overheads. Other compilers [2, 5, 31, 37, 65, 88] use communication optimization to generate efficient code for programs on distributed memory machines. Different ways of code generation result in different communication patterns at runtime. For example, the compiler may decide to send/receive all elements in an array to speed up the communication. It may also decide to send/receive one element at a time to save buffer space.

Communication optimizations reduce the cost of communication in a program. Communication performance not only affects the performance of a parallel application but also limits its scalability. Therefore, communication optimization is crucial for the performance of programs compiled for a distributed memory machine. Many communication optimizations are applied within a single loop using data dependence information. Examples of such optimizations include message vectorization [37, 88], collective communication [28, 51], message coalescing [37] and message pipelining [31, 37]. Earlier methods are based on *location based* data dependence, which is not precise since it only determines whether two references refer to the same memory location. Later schemes refine the information and use *value based* data dependence [2]. In value based data dependence, a read reference depends on a write reference only if the write provides the value for the read reference.

Communication optimizations based only on data dependence information usually result in redundant communications [14]. The more recently developed optimizations use data flow information to reduce redundant communication and perform other optimizations. In [24] a data flow framework which can integrate a number of communication optimizations is presented. However, the method can only apply to a very small subset of programs which are constrained in the forms of loop nests and array indices. In [32] a unified framework which uses global array data flow analysis for communication optimizations is described. Since only a very simplified version of the analysis algorithm is implemented, it is not clear whether this approach is practical for large programs. In [14, 41] methods that combine traditional data flow analysis techniques with data dependence analysis for performing global communication optimizations are described. These schemes are very efficient in terms of their analysis cost since bit vectors are used to represent data flow information. However, they cannot obtain the array data flow information that is as precise as the information computed using array data flow analysis approaches. Communication optimization changes the communication behavior of a program. Since many communication optimizations are commonly used in production compilers, these optimizations must be considered to obtain realistic communication patterns in a program.

Chapter 3

Dynamic single–hop communication

This chapter discusses the path reservation protocols for dynamic single–hop communication. Two types of distributed path reservation protocols, the *forward path reservation protocols* and the *backward path reservation protocols*, have been designed for point–to–point optical TDM networks. A network simulator that simulates all the protocols has been developed and has been used to study the performance of the two types of protocols and to evaluate the impact of system parameters such as the control packet processing time and the message size on the protocols.

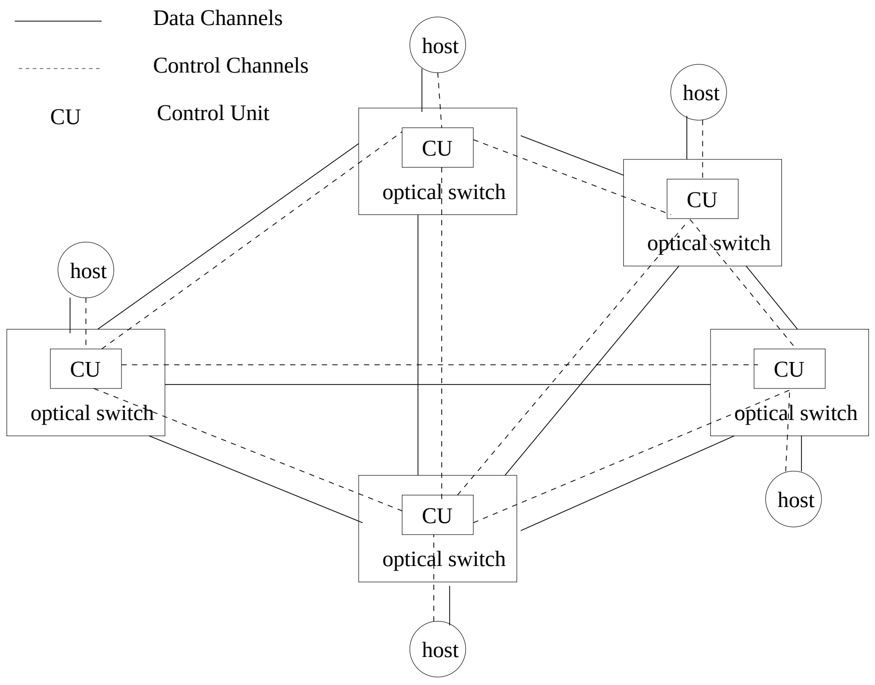


Figure 3.1: An optical network with distributed control.

In order to support a distributed control mechanism for connection establishment, it is assumed that in addition to the optical data network, there is a logical *shadow network* through which control messages are communicated. The shadow network has the same physical topology as the data network. The traffic on the shadow network consists of small control packets and thus is much lighter than the traffic on the data network. The shadow network operates in packet switching mode; routers at intermediate nodes examine control packets and update local bookkeeping information and switch states accordingly. The shadow network can be implemented as an electronic network or alternatively a virtual channel on the data network can be reserved exclusively for exchanging control messages. Figure 3.1 shows the network architecture. A virtual channel in the optical data network corresponds to a time slot. It is also assumed that a node can send or receive messages through different virtual channels simultaneously.

A path reservation protocol ensures that the path from a source node to a destination node is reserved before the connection is used. A path includes the virtual channels on the links that form the connection, the transmitter at the source node and the receiver at the destination node. Reserving the transmitter and the receiver is the same as reserving a virtual channel on the link from a node to the switch attached to that node. Hence, only the reservation of virtual channels on links forming a connection with path multiplexing will be considered. There are many options available with respect to different aspects of the path reservation mechanisms. These are discussed next.

- *Forward reservation* versus *backward reservation.* Locking mechanisms are needed by the distributed path reservation protocols to ensure the exclusive usage of a virtual channel for a connection. This variation characterizes the timing at which the protocols perform the locking. Under forward reservation, virtual channels are locked by a control message that travels from the source node to the destination node. Under backward reservation, a control message travels to the destination to probe the path, then virtual channels that are found to be available are locked by another control message which travels from the destination node to the source node.

- *Dropping* versus *holding.* This variation characterizes the behavior of the protocol when it determines that a connection establishment does not progress. Under the dropping approach, once the protocol determines that the establishment of a connection is not progressing, it releases the virtual channels locked on the partially established path and informs the source node that the reservation has failed. Under the holding approach, when the protocol determines that the establishment of a connection is not progressing, it keeps the virtual channels on the partially established path locked for some period of time, hoping that during this period, the reservation will progress. If, after this timeout period, the reservation still does not progress, the partial path is then released and the source node is informed of the failure. Dropping can be viewed as holding with holding time equal to zero.

- *Aggressive* reservation versus *conservative* reservation. This variation characterizes the protocol's treatment of each reservation. Under the aggressive reservation, the protocol tries to establish a connection by locking as many virtual channels as possible during the reservation process. Only one of the locked channels is then used for the connection, while the others are released. Under the conservative reservation approach, the protocol locks only one virtual channel during the reservation process.

Deadlock

Deadlock in the control network can arise from two sources. First, with limited number of buffers, a request loop can be formed within the control network. Second, deadlock can occur when a request is holding (locking) virtual channels on some links while requesting other channels on other links. This second source of deadlock can be avoided by the dropping or holding mechanisms described above. Specifically, a request will give up all the locked channels if it does not progress within a certain timeout period.

Many deadlock avoidance or deadlock prevention techniques for packet switching networks proposed in the literature [21] can be used to deal with deadlock within the control network (the first source of deadlock). Moreover, the control network is under light traffic, and each control message consists of only a single packet of small size (4 bytes). Hence, it is feasible to provide a large number of buffers in each router to reduce or eliminate the chances of deadlocks.

States of virtual channels

The control network router at each node maintains a state for each virtual channel on links connected to the router. For forward reservation, the control router maintains the states for the outgoing links. As discussed later, this enables the router to have the information needed for reserving virtual channels and updating the switch states. A virtual channel, V, on link L, can be in one of the following states:

- $AVAIL$: indicates that the virtual channel V on link L is available and can be used to establish a new connection,

- $LOCK$: indicates that V is locked by some request in the process of establishing a connection.

- $BUSY$: indicates that V is being used by some established connection to transmit data.

For a link, L, the set of virtual channels that are in the $AVAIL$ state is denoted as $Avail(L)$. When a virtual channel, V, is not in $Avail(L)$, an additional field, CID, is maintained to identify the connection request locking V, if V is in the $LOCK$ state, or the connection using V, if V is in the $BUSY$ state.

3.1 Forward reservation schemes

In the connection establishment protocols, each connection request is assigned a unique identifier, *id*, which consists of the identifier of the source node and a serial number issued by that node. Each control message related to the establishment of a connection carries its *id*, which becomes the identifier of the connection when it is successfully established. It is this *id* that is maintained in the CID field of locked or busy virtual channels on links. Four types of packets are used in the forward reservation protocols to establish a connection.

- *Reservation packets* (RES), used to reserve virtual channels. In addition to the connection *id*, a RES packet contains a bit vector, *cset*, of size equal to the number of virtual channels in each link. The bit vector *cset* is used to keep track of the set of virtual channels that can be used to satisfy the connection request carried by RES. These virtual channels are locked at intermediate nodes while the RES message progresses towards the destination node. The switch states are also set to connect the locked channels on the input and output links.

- *Acknowledgment packets* (ACK), used to inform source nodes of the success of connection requests. An ACK packet contains a *channel* field which indicates the virtual channel selected for the connection. As an ACK packet travels from the destination to the source, it changes the state of the virtual channel selected for the connection to $BUSY$, and unlocks (changes from $LOCK$ to $AVAIL$) all other virtual channels that were locked by the corresponding RES packet.

- *Fail or Negative ack packets* ($FAIL/NACK$), used to inform source nodes of the failure of connection requests. While traveling back to the source node, a $FAIL/NACK$ packet unlocks all virtual channels that were locked by the corresponding RES packet.

- *Release packets* (REL), used to release connections. A REL packet traveling from a source to a destination changes the state of the virtual channel reserved for that connection from $BUSY$ to $AVAIL$.

The protocols require that control packets from a destination, d, to a source, s, follow the same paths (in opposite directions) as packets from s to d. The fields of a packet will be denoted by $packet.field$. For example, $RES.id$ denotes the id field of the RES packet.

The forward reservation with dropping works as follows. When the source node wishes to establish a connection, it composes a RES packet with $RES.cset$ set to the virtual channels that the node may use. This message is then routed to the destination. When an intermediate node receives the RES packet, it determines the next outgoing link, L, on the path to the destination, and updates $RES.cset$ to $RES.cset \cap Avail(L)$. If the resulting $RES.cset$ is empty, the connection cannot be established and a $FAIL/NACK$ message is sent back to the source node. The source node will retransmit the request after some period of time. This process of failed reservation is shown in Figure 3.2(a). Note that if $Avail(L)$ is represented by a bit-vector, then $RES.cset \cap Avail(L)$ is a bit-wise "AND" operation.

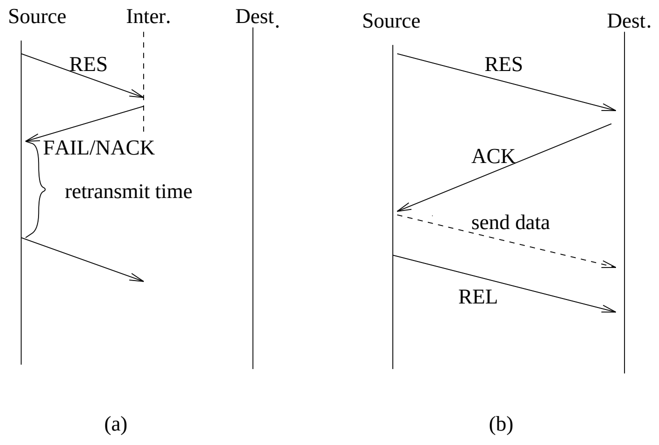


Figure 3.2: Control messages in forward reservation

If the resulting $RES.cset$ is not empty, the router reserves all the virtual channels in $RES.cset$ on link L by changing their states to $LOCK$ and updating $Avail(L)$. The router will then set the switch state to connect the virtual channels in the resulting $RES.cset$ of the corresponding incoming and outgoing links. Maintaining the states of outgoing links is sufficient for these two tasks. The RES message is then forwarded to the next node on the path to the destination. This way, as RES approaches the destination, the path is reserved incrementally. Once RES reaches the destination with a non-empty $RES.cset$, the destination selects from $RES.cset$ a virtual channel to be used for the connection and informs the source node that the channel is selected by sending an ACK message with $ACK.channel$ set to the selected virtual channel. The source can start sending data once it receives the ACK packet. After all data is sent, the source node sends a REL packet to tear down the connection. This successful reservation process is shown in Figure 3.2 (b). Note that although in the algorithm described above, the switches are set during the processing of the RES packet, they can instead be set during the processing of the ACK packet.

Holding: The protocol described above can be modified to use the holding policy instead of the dropping policy. Specifically, when an intermediate node determines that the connection for a reservation cannot be established, that is when $RES.cset \cap Avail(L) = \phi$, the node buffers the RES packet for a limited period of time. If within this period, some virtual channels in the original $RES.cset$ become available, the RES packet can then continue its journey. Otherwise, the $FAIL/NACK$ packet is sent back to the source. Implementing the holding policy requires each node to maintain a holding queue and to periodically check that queue to determine if any of the virtual channels has become available. In addition, some timing mechanism must be incorporated in the routers to timeout held control packets. This increases the hardware and software complexities of the routers.

Aggressiveness: The aggressiveness of the reservation is reflected in the size of the virtual channel set, $RES.cset$, initially chosen by the source node. In the most aggressive scheme, the source node sets $RES.cset$ to $\{0, ..., N-1\}$, where N is the number of virtual channels in the system. This ensures that the reservation will be successful if there exists an available virtual channel on the path. On the other hand, the most conservative reservation assigns $RES.cset$ to include only a single virtual channel. In this case, the reservation can be successful only when the virtual channel chosen by the source node is available in all the links on the path. Although the aggressive scheme seems to have advantage over the conservative scheme, it results in excessive locking of the virtual channels in the system. Thus, in heavily loaded networks, this is expected to decrease the overall throughput. To obtain optimal performance, the aggressiveness of the protocol should be chosen appropriately between the most aggressive and the most conservative extremes.

The retransmit time is another protocol parameter. In traditional non–multiplexed networks, the retransmit time is typically chosen randomly from a range [0,MRT], where MRT denotes some maximum retransmit time. In such systems, MRT must be set to a reasonably large value to avoid live-lock. However, this may increase the average message latency time and decrease the throughput. In a multiplexed network, the problem of live-lock only occurs in the most aggressive scheme (non–multiplexed circuit switching networks can be considered as having a multiplexing degree of 1 and using aggressive reservation).

For less aggressive schemes, the live-lock problem can be avoided by changing the virtual channels selected in *RES.cset* when RES is retransmitted. Hence, for these schemes, a small retransmit time can be used.

3.2 Backward reservation schemes

In the forward locking protocol, the initial decision concerning the virtual channels to be locked for a connection request is made in the source node without any information about network usage. The backward reservation scheme tries to overcome this handicap by probing the network before making the decision. In the backward reservation schemes, a forward message is used to probe the availability of virtual channels. After that, the locking of virtual channels is performed by a backward message. The backward reservation scheme uses six types of control packets, all of which carry the connection id, in addition to other fields as discussed next:

- *Probe packets* ($PROB$) travel from sources to destinations gathering information about virtual channel usage without locking any virtual channel. A $PROB$ packet carries a bit vector, $init$, to represent the set of virtual channels that are available to establish the connection.

- *Reservation packets* (RES) are similar to the RES packets in the forward scheme, except that they travel from destinations to sources, lock virtual channels as they go through intermediate nodes, and set the states of the switches accordingly. A RES packet contains a $cset$ field.

- *Acknowledgment packets* (ACK) are similar to ACK packets in the forward scheme except that they travel from sources to destinations. An ACK packet contains a $channel$ field.

- *Fail packets* ($FAIL$) unlock the virtual channels locked by the RES packets in cases of failures to establish connections.

- *Negative acknowledgment packets* ($NACK$) are used to inform the source nodes of reservation failures.

- *Release packets* (REL) are used to release connections after the communication is completed.

Note that a $FAIL/NACK$ message in the forward scheme performs the functions of both a $FAIL$ message and a $NACK$ message in the backward scheme.

The backward reservation with dropping works as follows. When the source node wishes to establish a connection, it composes a $PROB$ message with $PROB.init$ set to contain all virtual channels in the system. This message is then routed to the destination. When an intermediate node receives the $PROB$ packet, it determines the next outgoing link, L_f, on the forward path to the destination, and updates $PROB.init$ to $PROB.init \cap Avail(L_f)$. If the resulting $PROB.init$ is empty, the connection cannot be established and a $NACK$ packet is sent back to the source node. The source node will try the reservation again after a certain retransmit time. Figure 3.3(a) shows this failed reservation case.

If the resulting $PROB.init$ is not empty, the node forwards $PROB$ on L_f to the next node. This way, as $PROB$ approaches the destination, the virtual channels available on the path are recorded in the *init* set. Once $PROB$ reaches the destination, the destination forms a RES message with $RES.cset$ equal to a selected subset of $PROB.init$ and sends this message back to the source node. When an intermediate node receives the RES packet, it determines the next link, L_b, on the backward path to the source, and updates $RES.cset$ to $RES.cset \cap Avail(L_b)$. If the resulting $RES.cset$ is empty, the connection cannot be established. In this case the node sends a $NACK$ message to the source node to inform it of the failure, and sends a $FAIL$ message to the destination to free the virtual channels locked by RES. This process is shown in Figure 3.3 (b).

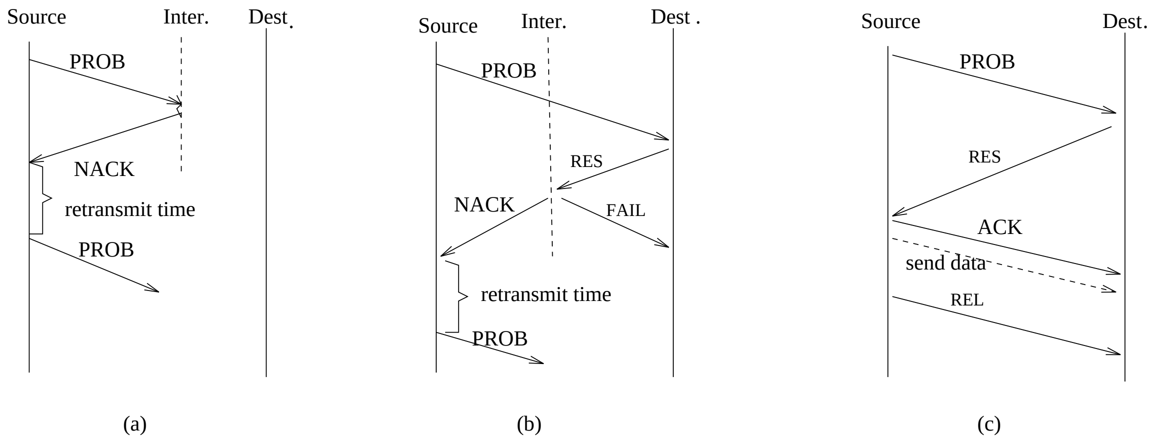


Figure 3.3: Control messages in backward reservation

If the resulting $RES.cset$ is not empty, the virtual channels in $RES.cset$ are locked, the switch is set accordingly and RES is forwarded on L_b to the next node. When RES reaches the source with a non-empty $RES.cset$, the source selects a virtual channel from the $RES.cset$ for the connection and sends an ACK message to the destination with $ACK.channel$ set to the selected virtual channel. This ACK message unlocks all the virtual channels locked by RES, except the one in *channel*. The source node can start sending data as soon as it sends the ACK message. After all data is sent, the source node sends a REL packet to tear down the connection. The process of successful reservation is shown in Figure 3.3(c).

Holding: Holding can be incorporated in the backward reservation scheme as follows. In the protocol, there are two cases that cause the reservation to fail. The protocol may determine that the reservation fails when processing the $PROB$ packet. In this case, holding is not desirable because the PROB packet is used to collect the channel usage information and holding could reduce the precision of the information collected (the status of channels on other links may change during the holding period). When the protocol determines that the reservation fails during the processing of a RES packet, a holding mechanism similar to the one used in the forward reservation scheme may be applied.

Aggressiveness: The aggressiveness of the backward reservation protocols is reflected in the initial size of *cset* chosen by the destination node. The aggressive approach sets $RES.cset$ equal to $PROB.init$, while the conservative approach sets $RES.cset$ to contain a single virtual channel from $PROB.init$. Note that if a protocol supports only the conservative scheme, the ACK messages may be omitted, and thus only five types of messages are needed. As in the forward reservation schemes, the retransmit time is a parameter in the backward schemes.

3.3 Network simulator and experimental results

A network simulator has been developed to simulate the behavior of multiplexed torus networks. The simulator models the network with various choices of system parameters and protocols. Specifically, the simulator provides the following options for protocol parameters.

- *forward and backward* reservations, this determines which protocol to be simulated.
- *initial cset size*: This parameter determines the initial size of *cset* in the reservation packet. It restricts the set of virtual channels under consideration for a reservation. In forward schemes, the initial *cset* is chosen when the source node composes the RES packet. Assuming that N is the multiplexing degree in the system, an $RES.cset$ of size s is chosen by generating a random number, m, in the range [0,N−1], and assigning $RES.cset = \{m\ mod\ N, m+1\ mod\ N..., N+s-1\ mod\ N\}$. In the backward schemes, the initial *cset* is set when the destination node composes the ACK packet. An $ACK.cset$ of size s is generated in the following manner. If the available set, $RES.INIT$, has less available channels than s, the $RES.INIT$ is copied to $ACK.cset$.

Otherwise, the available channels are represented in a linear array and the method used in generating the *cset* in the forward schemes is used.

- *timeout value*: This value determines how long a reservation packet can be put in a waiting queue. The dropping scheme can be viewed as a holding scheme with timeout time equal to zero.
- *maximum retransmit time* (MRT): This specifies the period after which a node will retry a failed reservation. As discussed earlier, this value is crucial for avoiding live-lock in the most aggressive schemes. The actual retransmit time is chosen randomly between 0 and $MRT - 1$.

Besides the protocol parameters, the simulator also allows the choices of various system parameters.

- *system size*: This specifies the size of the network. All our simulations are done on torus topology.
- *multiplexing degree.* This specifies the number of virtual channels supported by each link. In our simulation, the multiplexing degree ranges from 1 to 32.
- *message size*: The message size directly affects the time that a connection is kept before it is released. In our simulations, fixed size messages are assumed.
- *request generation rate at each node (r)*: This specifies the traffic on the network. The connection requests at each node are assumed to have a Poisson inter-arrival distribution. When a request is generated at a node, the destination of the request is generated randomly among the other nodes in the system. When a generated request is blocked, it is put into a queue, waiting to be re-transmitted.
- *control packet processing and propagation time*: This specifies the speed of the control networks. The control packet processing time is the time for an intermediate node to process a control packet. The control packet propagation time is the time for a control packet to be transferred from one node to the next. It is assumed that all the control packets have the same processing and propagation time.

In the following discussion, F is used to denote forward reservation, B denotes the backward reservation, H denotes holding and D denotes dropping schemes. For example, FH means the forward holding scheme. In addition to the options of backward/forward reservation and holding/dropping policy, the simulation uses the following parameters. The average latency and throughput are used to evaluate the protocols. The latency is the period between the time when a message is ready and the time when the first packet of the message is sent. The throughput is the number of messages received per time unit. Under light traffic, the performance of the protocols is measured by the average message latency, while under heavy traffic, the throughput is used as the performance metric. The simulation time is measured in time slots, where a time slot is the time to transmit an optical data packet between any two nodes in the network. Note that in multiprocessor applications, nodes are physically close to each other, and thus signal propagation time is very small (1 foot per nsec) compared to the length of a message. Finally, deterministic XY–routing is assumed in the torus topology.

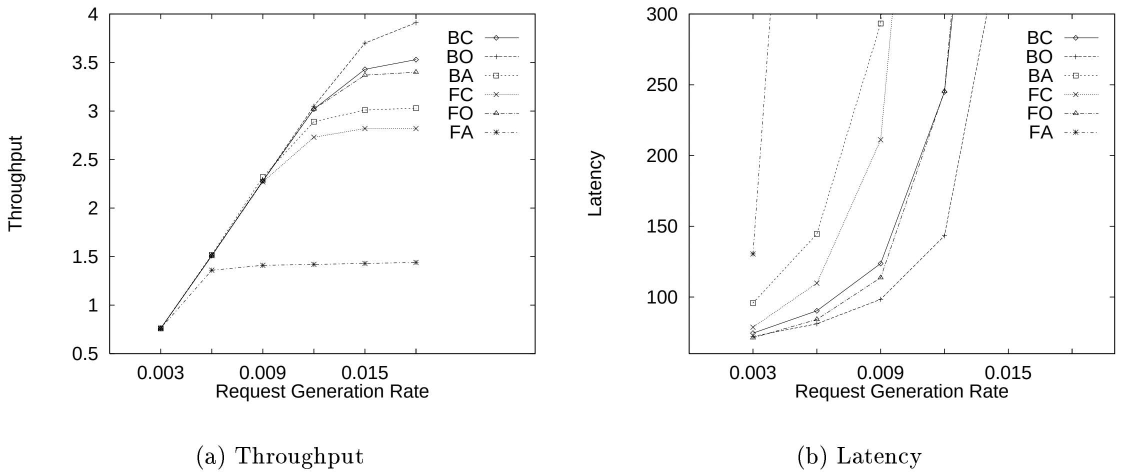


Figure 3.4: Comparison of the reservation schemes with dropping

Figure 3.4 depicts the throughput and average latency as a function of the request generation rate for six protocols that use the dropping policy in a 16×16 torus. The multiplexing degree is taken to be 32, the message size is assumed to be 8 packets and the control packets processing and propagation time is assumed to be 2 time units. For each of the forward and backward schemes, three variations are considered with varying aggressiveness. The conservative variation in which the initial *cset* size is 1, the most aggressive variation in which the initial set size is equal to the multiplexing degree and an optimal variation in which the initial set size is chosen (by repeated trials) to maximize the throughput. The letters C, A and O are used to denote these three variations, respectively. For example, FDO means the forward dropping scheme with optimal *cset* size. Note that the use of the optimal *cset* size reduces the delay in addition to increasing the throughput. Note also that the network saturates when the generation rate is between 0.006 and 0.018, depending on the protocol used. The maximum saturation rate that the 16×16 torus can achieve in the absence of contention and control overhead is given by

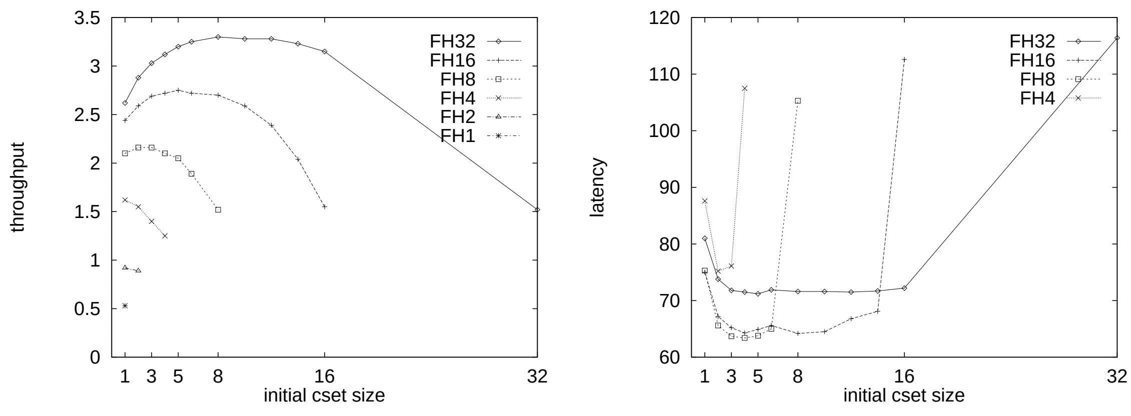
$$\frac{number\ of\ links}{no.\ of\ PEs \times av.\ no.\ of\ links\ per\ msg \times msg\ size} = \frac{1024}{256 \times 8 \times 8} = 0.0625.$$

Hence, the optimal backward protocol can achieve almost 30% of the theoretical full utilization rate.

Figure 3.4(b) also reveals that, when the request generation rate, r, is small, for example $r = 0.003$, the network is under light traffic and all the protocols achieve the same throughput, which is equal to r times the number of processors. In this case, the performance of the network should be measured by the average latency. In the rest of the performance study, the maximum throughput (at saturation) and the average latency (at $r = 0.003$) were used to measure the performance of the protocols. Two sets of experiments are performed. The first set evaluates the effect of the protocol parameters on the network throughput and delay and the second set evaluates the impact of system parameters on performance.

Effect of protocol parameters

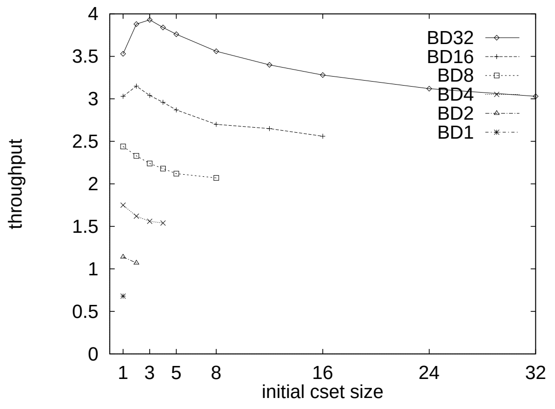
In this set of experiments, the effect of the initial *cset* size, the holding time and the retransmit time on the performance of the protocols are studied. The system parameters for this set of experiments are chosen as follows: system size = 16×16, message size = 8 packets, control packet processing and propagation time = 2 time units.



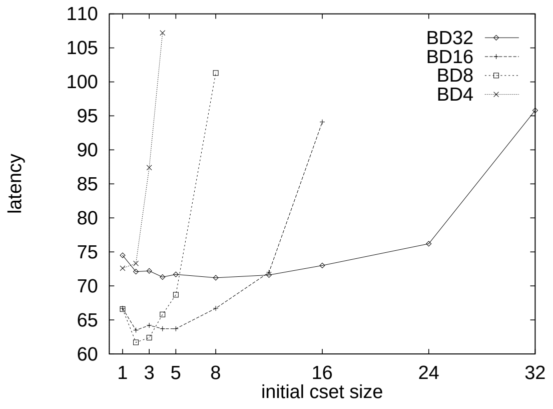
(a) Maximum Throughput (b) Latency

Figure 3.5: Effect of the initial *cset* size on forward schemes

Figure 3.5 shows the effect of the initial *cset* size on the forward holding scheme with different multiplexing degrees, namely 1, 2, 4, 8, 16 and 32. The holding time is taken to be 10 time units and the MRT is 5 time units for all the protocols with initial *cset*



(a) Maximum Throughput



(b) Latency

Figure 3.6: Effect of the initial *cset* size on backward schemes

size less than the multiplexing degree and 60 time units for the most aggressive forward scheme. Large MRT is used in the most aggressive forward scheme because it is observed that small MRT often leads to live-lock in this scheme. Only the protocols with the holding policy will be shown since using the dropping policy leads to similar patterns. The effect of holding/dropping will be considered in a later figure. Figure 3.6 shows the results for the backward schemes with the dropping policy.

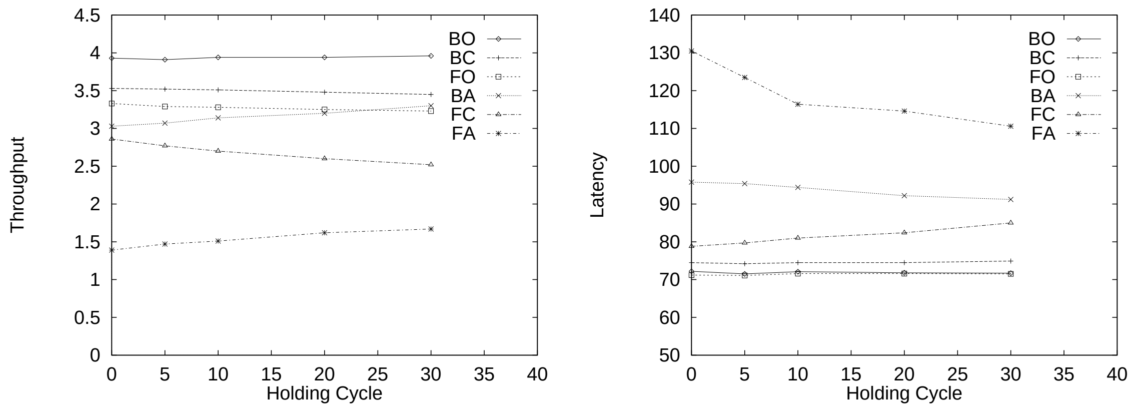
From Figure 3.5 (a), it can be seen that when the multiplexing degree is larger than 8, both the most conservative protocol and the most aggressive protocol do not achieve the best throughput. Figure 3.5(b) shows that these two extreme protocols do not achieve the smallest latency either. The same observation applies to the backward schemes in Figure 3.6. The effect of choosing the optimal initial *cset* is significant on both throughput and delay. That effect, however, is more significant in the forward scheme than in the backward scheme. For example, with multiplexing degree = 32, choosing a non-optimal *cset* size may reduce the throughput by 50% in the forward scheme and only by 25% in the backward scheme. In general, the optimal initial *cset* size is hard to find. Table 3.1 lists the optimal initial *cset* size for each multiplexing degree. A rule of thumb to approximate the optimal *cset* size is to use 1/3 and 1/10 of the multiplexing degree for forward schemes and backward schemes, respectively.

Figure 3.7 shows the effect of the holding time on the performance of the protocols for a multiplexing degree of 32. As shown in Figure 3.7(a), the holding time has little effect on the maximum throughput. It slightly increases the performance for the forward aggressive and the backward aggressive schemes. As for the average latency under light work load, the holding time also has little effect except for the forward aggressive scheme,

Multiplexing Degree	Optimal *cset* size	
	Forward	Backward
4	1	1
8	2	1
16	5	2
32	10	3

Table 3.1: Optimal *cset* size

where the latency time decreases by about 20% when the holding time at each intermediate node increases from 0 to 30 time units. Since holding requires extra hardware support compared to dropping, it is concluded that holding is not cost–effective for the reservation protocols. In the rest of the paper, only protocols with dropping policies will be considered.



(a) Maximum Throughput (b) Latency

Figure 3.7: Effect of holding time

Figure 3.8 shows the effect of the maximum retransmit time on the performance. Note that the retransmit time is uniformly distributed in the range $0..MRT-1$. As shown in Figure 3.8 (a), increasing MRT results in performance degradation in all the schemes except FDA, in which the performance improves with MRT. This confirms that the MRT value is important to avoid live-lock in the network when aggressive reservation is used. In other schemes this parameter is not important, because when retransmitting a failed request, virtual channels different than the ones that have been tried may be included in *cset*. This result indicates another drawback of the forward aggressive schemes: in order to avoid live-lock, the MRT must be a reasonably large value, which decreases the overall performance.

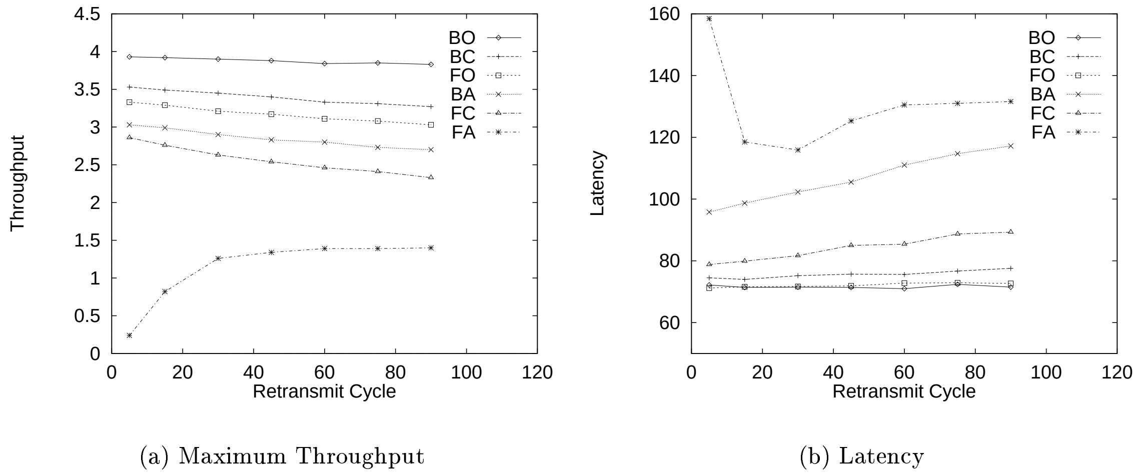


Figure 3.8: Effect of maximum retransmit time

Effect of other system parameters

In this section, only dropping schemes with MRT equal to 5 time units for all schemes except FDA will be considered. The MRT for FDA schemes is set to 60. This set of experiments focuses on studying the performance of the protocols under different multiplexing degrees, system sizes, message sizes and control network speeds. One parameter is changed in each experiment, with the other parameters set to the following default values (unless stated otherwise): network size = 16×16 torus, multiplexing degree = 16, message size = 8 packets, control packet processing and propagation time = 2 time units.

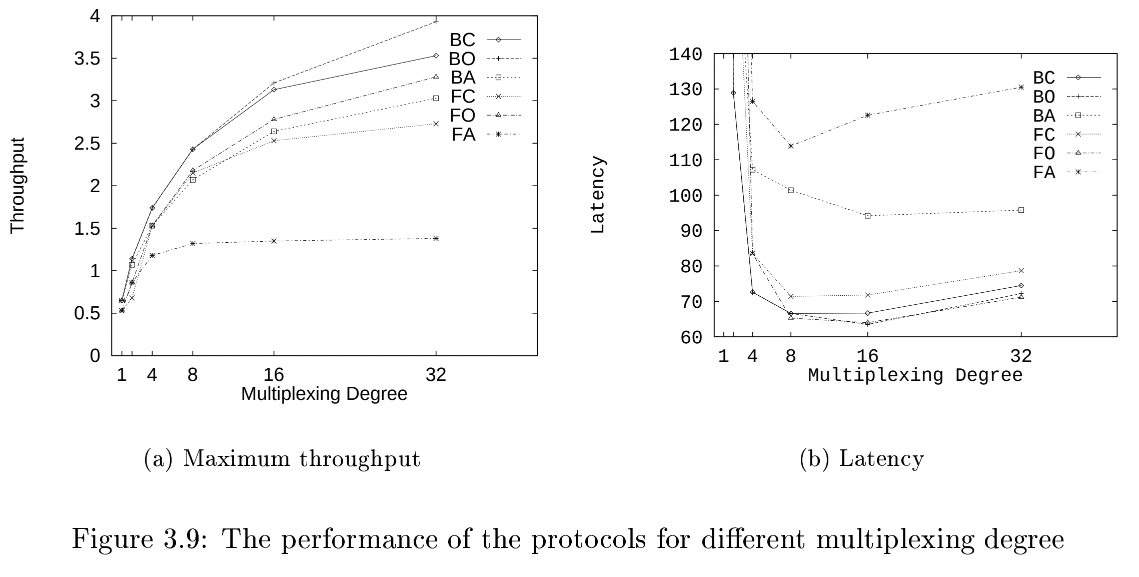
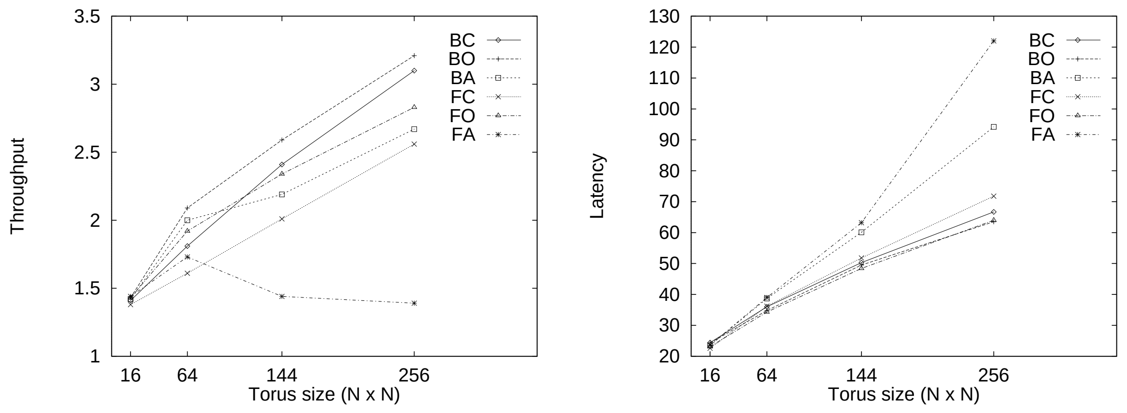


Figure 3.9: The performance of the protocols for different multiplexing degree

Figure 3.9 shows the performance of the protocols for different multiplexing degrees. When the multiplexing degree is small, BDO and FDO have the same maximum bandwidth as BDC and FDC, respectively. When the multiplexing degree is large, BDO and FDO offer better throughput. In addition, for all multiplexing degrees, BDO is the best among all the schemes. As for the average latency, both FDA and BDA have significantly larger latency than all other schemes. Also, FDO and BDO have the smallest latencies. It can be seen from this experiment that the backward schemes always provide the same or better performance (both maximum throughput and latency) than their forward reservation counterparts for all multiplexing degrees considered.

Figure 3.10 shows the effect of the network size on the performance of the protocols. It can be seen from the figure that all the protocols, except the aggressive ones, scale nicely with the network size. This indicates that the aggressive protocols cannot take advantage of the spatial diversity of the communication. This is a result of excessive reservation of channels. When the network size is small, there is little difference in the performance of the protocols. When the network size is larger, the backward schemes show their superiority.



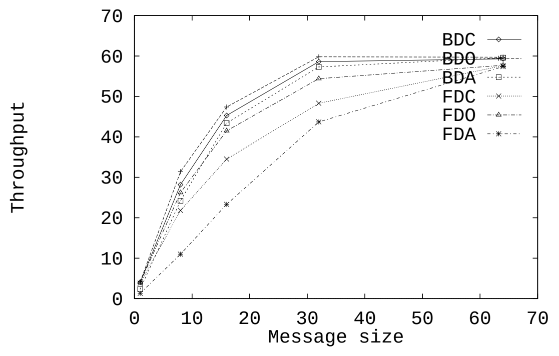
(a) Maximum throughput (b) Latency

Figure 3.10: Effect of the network size

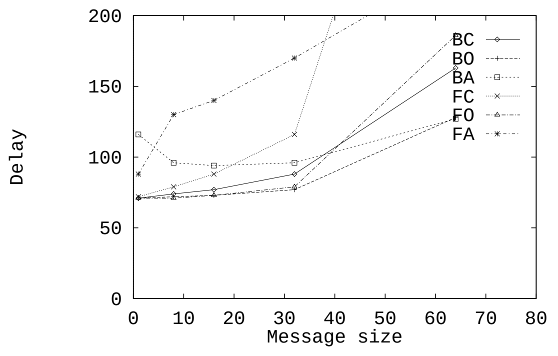
Figure 3.11 shows the effect of the message size on the protocols. The multiplexing degree in this experiment is 16. The throughput in this figure is normalized to reflect the number of packets that pass through the network, rather than the number of messages, that is,

$$normalized\ throughput \;=\; msg\ size \times\ throughput.$$

Both the forward and backward locking schemes achieve higher throughput for larger messages. When messages are sufficiently large, the signaling overhead in the protocols is small and all protocols have almost the same performance. However, when the message size is small, the BDO scheme achieves higher throughput than the other schemes. This indicates that BDO incurs less overhead in the path reservation than the other schemes.



(a) Maximum throughput

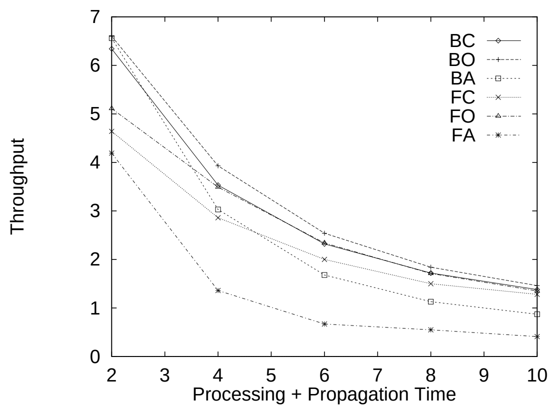


(b) Latency

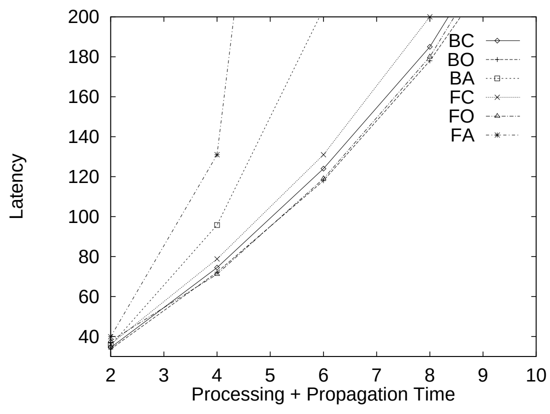
Figure 3.11: Effect of the message size

The effect of message size on the latency of the protocols is interesting. Forward schemes incur larger latency when the message size is large. By blindly choosing initial cset, forward schemes do not avoid choosing virtual channels used in communications, which increases the latency when the message size is large (so that connections are held longer for communications). Backward schemes probe the network before choosing the initial csets. Hence, the latency in backward schemes does not increase as much as in forward schemes when message size increases. Another observation is that in both forward and backward protocols, aggressive schemes sustain the increment of message size better than the conservative schemes. This is also because of the longer communication time with larger message sizes. Aggressive schemes are more efficient in finding a path in case of large message size. Note that this merit of aggressive schemes is offset by over reservation. Another interesting point is that the latency for messages of size 1 results in higher latency than messages of size 8 in BDA scheme. This can be attributed to too many control messages in the network in the case when data message contains a single packet (and thus can be transmitted fast). The conflicts of control messages result in larger latency.

Figure 3.12 shows the effect of the control network speed on performance. The multiplexing degree in this experiment is 32. The speed of the control network is determined by the time for a control packet to be transferred from one node to the next node and the time for the control router to process the control packet. From the figure it can be seen that when the control speed is slower, the maximum throughput and the average latency degrade. The most aggressive schemes in both forward and backward reservations, however, are more sensitive to the control network speed. Hence, it is important to have a reasonably fast control network when these reservation protocols are used.



(a) Maximum Throughput



(b) Latency

Figure 3.12: Effect of the speed of the control network

3.4 Chapter summary

This chapter presented the forward path reservation algorithms and the backward path reservation algorithms to establish connections with path multiplexing for connection requests that arrive at the network dynamically. Holding and dropping variants of these protocols were described. A holding scheme holds the reservation packet for a period of time when it determines that there is no available channel on the next link for the connection. A dropping scheme drops the reservation packet and starts a new reservation once it finds that there is no available channel on the next link for the connection. Protocols with various aggressiveness were discussed. The most aggressive schemes reserve as many channels as possible for each reservation to increase the probability of a successful reservation, while the most conservative schemes reserve one channel for each reservation to reduce the over–locking problem. The performance of the protocols and the impact of system parameters on the protocols were studied. The major results obtained in the experiments are summarized as follows.

- With proper protocols, multiplexing results in higher maximum throughput. Multiplexed networks are significantly more efficient than non–multiplexed networks.
- Both the most aggressive and the most conservative reservations cannot achieve optimal performance. The performance of the forward schemes is more sensitive to the aggressiveness than the performance of the backward schemes.
- The value of the holding time in the holding schemes does not have significant impact on the performance. In general, however, dropping is more efficient than holding.

- The retransmit time has little impact on all the schemes except the forward aggressive dropping scheme.
- The performance of the forward aggressive dropping scheme is significantly worse than other protocols. Moreover, this protocol cannot take advantage of both larger multiplexing degree and larger network size.
- The backward reservation schemes provide better performance than the forward reservation schemes for all multiplexing degrees.
- The difference of the protocols does not affect the communication efficiency when the network size is small. However, for large networks, the backward schemes provide better performance.
- The backward schemes provide better performance when the message size is small. When the message size is large, all the protocols have similar performance.
- The speed of the control network significantly affects the performance of the protocols.

These protocols achieve all–optical communication in data transmission. However, they require extra hardware support to exchange control messages and incur large startup overhead. An alternative to the single–hop communication is the multi–hop communication. Multi–hop networks do not require extra hardware support. They use intermediate hops to route messages toward their destinations. In the next chapter, multi–hop networks are considered.

Chapter 4
Dynamic multi–hop communication

Since by using time–division multiplexing multiple channels are supported on an optical link, more sophisticated logical topologies can be realized on top of a simpler physical network to improve the communication performance. These logical topologies reduce the number of intermediate hops that a packet travels at the cost of a larger multiplexing degree. On the one hand, the large multiplexing degree increases the packet communication time between hops. On the other hand, reducing the number of intermediate hops reduces the time spent at intermediate nodes. This chapter studies the trade–off between the multiplexing degree and the number of intermediate hops needed in logical topologies implemented on top of physical torus networks. Specifically, four logical topologies ranging from the most complex logical all–to–all connections to the simplest logical torus topology are examined. An analytical model for the maximum throughput and the average packet delay is developed and verified through simulations. The performance and the impact of system parameters on the performance for these four topologies are studied. Furthermore, the performance of the multi–hop communication using an efficient logical topology is compared with that of the single–hop communication using a distributed path reservation protocol, and the advantages and the drawbacks of these two communication schemes are identified.

To perform multi–hop communication, packets may be routed through intermediate nodes. Specifically, a communication module at each node, which will be referred to as the *router* in this chapter, is needed to route packets toward their destinations. It is assumed that each router contains a *routing buffer* that buffers all incoming packets. For each packet, the router determines whether to deliver the packet to the local PE or to the next link toward the packet destination. A separate *output path buffer* is used for each outgoing path that buffers the packets to be sent on that path and thus accommodates the speed mismatch between the electronic router and the optical path. Figure 4.1 depicts the structure of a router (see also Figure 2.5). Note that the output paths are multiplexed in time over the physical links that connect the local PE to its corresponding switch. In the rest of the chapter, *routing delay* will be used to denote the time a packet spends in the routing buffer and the time for the router to make a routing decision for the packet (packet routing time). *Transmission delay* will be used to denote the time a packet spends on the path buffer and the time it takes for the packet to be transferred on the path.

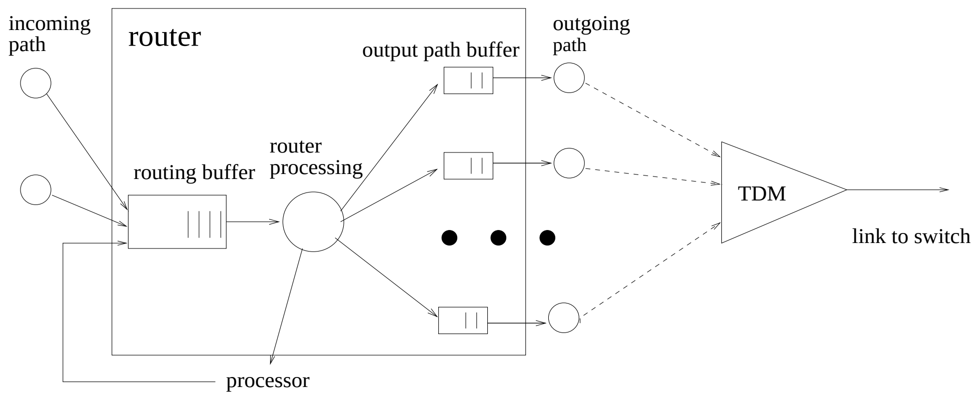


Figure 4.1: A router

4.1 Realizing logical topologies on physical torus topology

Four logical topologies are considered in this section, the logical torus topology, the logical hypercube topology, the logical all–to–all topology, and the logical allXY topology where all–to–all connections are established along each dimension. Let us consider an example in which a packet is transmitted from node 0 to node 11 in the 4×4 torus shown in Figure 4.2. Using the logical all–to–all topology, the packet will go directly from node 0 to node 11. Using the logical allXY topology, the packet will go from node 0 to node 3 to node 11. Using the logical hypercube topology, the packet will go from node 0(0000) to node 1(0001) to node 3(0011) to node 11(1011). Using the logical torus topology, the packet will go from node 0 to node 3 to node 7 to node 11.

Traditional embedding techniques that minimize the *congestion* for a given communication pattern are not adequate for minimizing the number of virtual channels needed to realize the communication in an optical network with path multiplexing. The congestion is usually not equal to the number of channels needed to realize a communication pattern. Consider the example in Figure 4.3 in which the congestion in the network is 2, while 3 channels are needed to realize the three connections. To efficiently realize a logical topology in an optical network, both routing and channel assignment (RCA) options must be taken into consideration. Schemes to realize these four logical topologies on the physical torus topology will be discussed next.

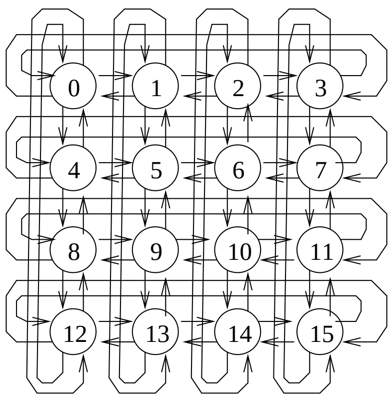


Figure 4.2: Node numbering in a torus topology

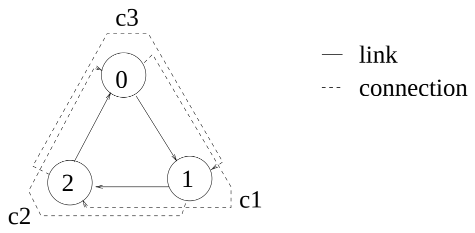


Figure 4.3: Difference between embedding and RCA

4.1.1 Logical hypercube topology

This subsection considers the optimal schemes to realize the logical hypercube topology on top of the physical torus topologies. Since the algorithm to realize the hypercube topology on the physical torus topologies utilizes the algorithms to realize the hypercube on top of physical mesh, ring and array topologies, algorithms to realize the hypercube topology on top of all these mesh–like topologies are discussed.

Given networks of size N, it will be proven that $\lfloor \frac{2N}{3} \rfloor$ and $\lfloor \frac{N}{3} + \frac{N}{4} \rfloor$ channels are the minimum required to realize hypercube communication on array and ring topologies, respectively. Routing and channel assignment schemes that achieve these minimum requirements are developed, indicating that the bounds are tight and the schemes are optimal. These schemes are extended to mesh and torus topologies and it is proven that for a $2^k \times 2^{r-k}$ $(k \geq r-k)$ mesh or torus, $\lfloor \frac{2 \times 2^k}{3} \rfloor$ and $\lfloor \frac{2^k}{3} + \frac{2^k}{4} \rfloor$ channels are the minimum required for realizing hypercube communication on these two topologies, respectively. Routing and channel assignment schemes are designed that use at most two more channels than the optimal to realize hypercube communication on these topologies. In the following sections, first the problem of routing and channel assignments for the hypercube communication on the physical mesh–like topologies is formally defined, and then the algorithms are described.

4.1.1.1 Problem definition

A network is modeled as a directed graph G(V, E), where nodes in V are switches and edges in E are links. Each node in a network is assigned a node number starting from 0. It is assumed that in arrays and rings the nodes are numbered from left to right in ascending order, and that the nodes are numbered in row major order for meshes and tori of size $n \times m$. Thus, the node in the ith column and the jth row is numbered as $j \times m + i$. This subsection focuses on studying the optimal RCA schemes for these traditional numbering schemes. Optimal node numbering (and its RCA) is a much more complex problem and is not considered in this dissertation. The number of nodes in a network is assumed to be $N = 2^r$. For a mesh or a torus to contain 2^r nodes, each row and column must contain a number of nodes that is a power of two. Hence, the size of meshes and tori is denoted as $N = 2^k \times 2^{r-k}$. Without losing generality, it is always assumed that $k \geq r - k$. The notations $ARRAY(N)$ and $RING(N)$ are used to represent arrays and rings of size N respectively, and $MESH(2^k \times 2^{r-k})$ and $TORUS(2^k \times 2^{r-k})$ for meshes and tori of size $2^k \times 2^{r-k}$ respectively.

The connection from node src to node dst is denoted as (src, dst). A *communication pattern* is a collection of connections. The *hypercube* communication pattern contains a connection (src, dst) if and only if the binary representations of src and dst differ in precisely one bit. A connection in the hypercube communication pattern is called a *dimension l connection* if it connects two nodes that differ in the lth bit position. In a network of size $N = 2^r$, the set, DIM_l, where $0 \leq l \leq r - 1$, is defined as the set of all dimension l connections and H_r is defined as the hypercube communication pattern. That is

$$DIM_l = \{(i, i + 2^l) \mid i \textit{ mod } 2^{l+1} < 2^l\} \cup \{(i, i - 2^l) \mid i \textit{ mod } 2^{l+1} \geq 2^l\}$$

$$H_r = \cup_{l=0}^{r-1} DIM_l$$

It can be easily proven that removing any DIM_l, for any $l \leq r - 1$, from H_r leaves two disjoint sets of connections, each of which being a hypercube pattern on $\frac{N}{2}$ nodes. For example, removing DIM_0 from H_r results in an H_{r-1} on the even–numbered nodes and another H_{r-1} on the odd–numbered nodes once the nodes are properly renumbered. Next, some definitions are introduced and the results of this section are summarized.

Definition: $P(x, y)$ is a *directed path* in G from node x to node y. It consists of a set of consecutive edges beginning at x and ending at y.

Definition: Given a network G and a communication pattern I, a *routing R(I)* of I is a set of directed paths $R(I) = \{P(x, y) | (x, y) \in I\}$.

Definition: Given a network G, a communication pattern I and a routing R(I) for the communication pattern, the congestion of an edge $\alpha \in E$, denoted as $\pi(G, I, R(I), \alpha)$, is the number of paths in R(I) containing α. The *congestion* of G in the routing R(I), denoted as $\pi(G, I, R(I))$, is the maximum congestion of any edge of G in the routing R(I), that is, $\pi(G, I, R(I)) = max_\alpha\{\pi(G, I, R(I), \alpha)\}$. The *congestion* of G for a communication pattern I, denoted as $\pi(G, I)$, is the minimum congestion of G in any routing R(I) for I, that is, $\pi(G, I) = min_R\{\pi(G, I, R(I))\}$.

Definition: Given a network G and a routing $R(I)$ for communication pattern I, an *assignment function* $A : R \to INT$, is a mapping from the set of paths to the set of integers INT, where an integer corresponds to a channel. A *channel assignment* for a routing $R(I)$ is an assignment function A that satisfies the following conditions:

1. If $P(x_1, y_1)$, $P(x_2, y_2)$ are different paths that share a common edge, then $A(P(x_1, y_1)) \neq A(P(x_2, y_2))$. This condition ensures that each channel on one link can only be assigned to one connection (i.e., there are no link conflicts).
2. $A(P(x, y_1)) \neq A(P(x, y_2))$ and $A(P(x_1, y)) \neq A(P(x_2, y))$. This condition ensures that each node can only use one channel at a time to send to or receive from other nodes (i.e., there are no node conflicts).

$A(R)$ denotes the set of channels assigned to the paths in R and $|A(R)|$ is the size of $A(R)$. Let $w(G, I, R)$ denote the minimum number of channels for the routing R, that is, $w(G, I, R) = min_A\{|A(R)|\}$. $w(G, I)$ denotes the smallest $w(G, I, R)$ over all R, i.e. $w(G, I) = min_R\{w(G, I, R)\}$

Lemma 1: $w(G, I) \geq \pi(G, I)$.

Proof: Follows directly from the above definitions. □

The following sections show that

$$w(ARRAY(N), H_r) = \pi(ARRAY(N), H_r) = \lfloor \frac{2N}{3} \rfloor$$

$$w(RING(N), H_r) = \pi(RING(N), H_r) = \lfloor \frac{N}{3} + \frac{N}{4} \rfloor$$

$$w(MESH(2^k \times 2^{r-k}), H_r) \leq \lfloor \frac{2 \times 2^k}{3} \rfloor + 2 \leq \pi(MESH(2^k \times 2^{r-k}), H_r) + 2$$

$$w(TORUS(2^k \times 2^{r-k}), H_r) \leq \lfloor \frac{2^k}{3} + \frac{2^k}{4} \rfloor + 2 \leq \pi(TORUS(2^k \times 2^{r-k}), H_r) + 2$$

4.1.1.2 Hypercube on linear array

Since routing in a linear array is fixed, the RCA problem is reduced to a channel assignment problem. Given a linear array of size $N = 2^r$, it is proven that $\lfloor \frac{2N}{3} \rfloor$ channels is the lower bound for realizing the hypercube communication by showing that $\pi(ARRAY(N), H_r) \geq \lfloor \frac{2N}{3} \rfloor$. A channel assignment scheme is developed that uses $\lfloor \frac{2N}{3} \rfloor$ channels for the hypercube communication. This proves that the bound is a tight lower bound and that the channel assignment scheme is optimal.

A lower bound

Using Lemma 1, a lower bound is obtained by proving that there exists a link in the linear array that is used $\lfloor \frac{2N}{3} \rfloor$ times when realizing H_r. The following lemmas establish the bound.

Lemma 2: In a linear array of size $N = 2^r$, where $r \geq 2$, there are 2^{r-1} connections in $DIM_{r-1} \cup DIM_{r-2}$ that use the link $(n, n+1)$ for any specific n satisfying $2^{r-2} \leq n \leq 2^{r-1} - 1$.

Proof: The connections in DIM_{r-1} and DIM_{r-2} can be represented by

$$DIM_{r-1}=\{(i, i+\tfrac{N}{2})|0 \leq i < \tfrac{N}{2}\} \cup \{(i, i-\tfrac{N}{2})|\tfrac{N}{2} \leq i < N\}$$

$$\begin{aligned} DIM_{r-2} = & \{(i, i+\tfrac{N}{4})|0 \leq i < \tfrac{N}{4} \text{ or } \tfrac{N}{2} \leq i < \tfrac{3N}{4}\} \\ \cup & \{(i, i-\tfrac{N}{4})|\tfrac{N}{4} \leq i < \tfrac{N}{2} \text{ or } \tfrac{3N}{4} \leq i < N\} \end{aligned}$$

Consider the connections in DIM_{r-1}. All connections $(i, i+\frac{N}{2})$ with $0 \leq i \leq n$ use link $(n, n+1)$, where $2^{r-2} \leq n \leq 2^{r-1} - 1$. Hence, as shown in Fig. 4.4 (a), there are n+1 connections in DIM_{r-1} that use link $(n, n+1)$. Similarly, in DIM_{r-2}, all connections $(i, i+\frac{N}{4})$, where $n < i + \frac{N}{4} < \frac{N}{2}$, use link $(n, n+1)$. As shown in Fig. 4.4 (b), there are $2^{r-1} - n - 1$ such connections. Hence, there are a total of $n + 1 + 2^{r-1} - n - 1 = 2^{r-1}$ connections in DIM_{r-1} and DIM_{r-2} that use link $(n, n+1)$. □

Lemma 3: In a linear array of size $N = 2^r$, there exists a link $(n, n+1)$ such that at least $\lfloor \frac{2N}{3} \rfloor$ connections in H_r use that link.

Proof: Let $T_i(2^r)$ be the number of connections in H_r that use link $(i, i+1)$ and let $T(2^r) = max_i(T_i(2^r))$. Thus $T(2^0) = 0$ and $T(2^1) = 1$. From Lemma 2, one knows that for $2^{r-2} \leq n \leq 2^{r-1} - 1$, link $(n, n+1)$ is used 2^{r-1} times by connections in DIM_{r-1} and DIM_{r-2}. Thus, the links in the second quarter of the array (from node 2^{r-2} to node $2^{r-1} - 1$) are used 2^{r-1} times by dimension $r-1$ and dimension $r-2$ connections. By the definition of hypercube communication, it is known that dimension 0 to dimension $r-3$

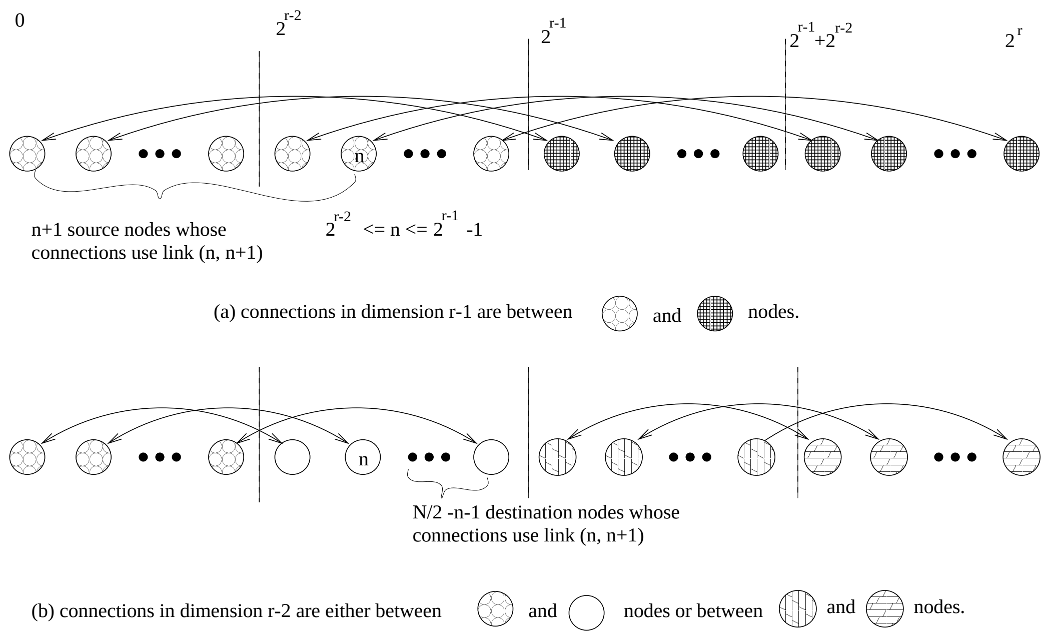

0
2^r-2
2^r-1
2^r-1 +2^r-2
2^r
n
n+1 source nodes whose connections use link (n, n+1)
2^r-2 <= n <= 2^r-1 -1
(a) connections in dimension r-1 are between and nodes.
n
N/2 -n-1 destination nodes whose connections use link (n, n+1)
(b) connections in dimension r-2 are either between and nodes or between and nodes.

Figure 4.4: Dimension $r-1$ and $r-2$ connections

connections form a hypercube on this quarter of the array. Thus, Lemma 2 can be recursively applied and the following inequality is obtained.

$$T(2^r) \geq 2^{r-1} + T(2^{r-2})$$

It can be proven by induction that the above inequality and the boundary conditions $T(2^0) = 0$, $T(2^1) = 1$, imply that $T(N) = T(2^r) \geq \lfloor \frac{2N}{3} \rfloor$. Hence, there exists a link which is used at least $\lfloor \frac{2N}{3} \rfloor$ times by connections in H_r. $\Box$

The proof of Lemma 3 is constructive in the sense that the link that is used at least $\lfloor \frac{2N}{3} \rfloor$ times can be found. By recursively considering the second quarter of the linear array, one can conclude that the source node, n, of the link $(n, n+1)$ that is used at least $\lfloor \frac{2N}{3} \rfloor$ times in H_r is $n = \frac{N}{4} + \frac{N}{16} + \frac{N}{64} + .. = \lfloor \frac{N}{3} \rfloor$. Hence, the link that is used at least $\lfloor \frac{2N}{3} \rfloor$ times in H_r is $(\lfloor \frac{N}{3} \rfloor, \lceil \frac{N}{3} \rceil)$.

Corollary 3.1 Give an array of size $N = 2^r$, if the nodes in the array are partitioned into 2 sets $S_1 = \{i | 0 \leq i \leq n\}$ and $S_2 = \{i | n+1 \leq i \leq N\}$, where $n = \lfloor \frac{N}{3} \rfloor$, then there are at least $\lfloor \frac{2N}{3} \rfloor$ connections in H_r from S_1 to S_2 and $\lfloor \frac{2N}{3} \rfloor$ connections from S_2 to S_1. $\Box$

Theorem 1: $\pi(ARRAY(N), H_r) \geq \lfloor \frac{2N}{3} \rfloor$.

Proof: Directly from Lemma 3. $\Box$

An optimal channel assignment scheme

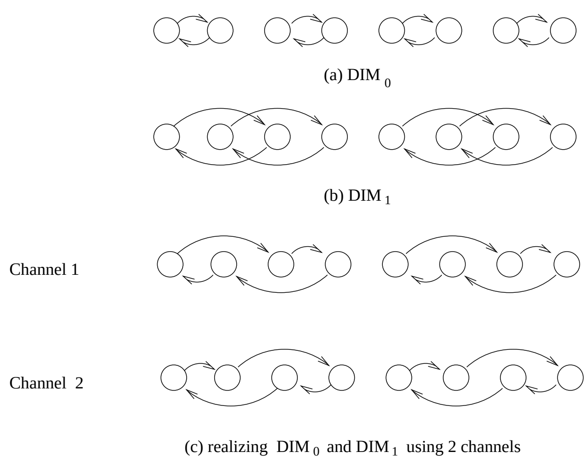


Figure 4.5: Realizing $DIM_0 \cup DIM_1$ of H_3

By the definition of hypercube communication, connections in H_r can be partitioned into three sets, DIM_0, $EVEN_r$ and ODD_r. DIM_0 contains the dimension 0 connections, $EVEN_r$ contains connections between nodes with even node numbers, and ODD_r contains connections between nodes with odd node numbers. Each of $EVEN_r$ and ODD_r forms a $r-1$ dimensional hypercube communication, H_{r-1}, if only the nodes involved in communications are considered and that the nodes are renumbered accordingly. Thus, channel assignment schemes for H_{r-1} can be extended to realize H_r as shown in the following lemma.

Lemma 4: Assuming that H_{r-1} can be realized on an array of size 2^{r-1} using K channels, then H_r can be realized on an array of size 2^r using $2K+1$ channels.

Proof: $H_r = EVEN_r \cup ODD_r \cup DIM_0$. From the above discussion and the assumption, $EVEN_r$ and ODD_r are H_{r-1} (when nodes are properly renumbered), K channels can be used to realize $EVEN_r$ or ODD_r. Since it can be easily proven that DIM_0 can be realized with one channel, a total of $2K+1$ channels can be used to realize H_r. $\Box$

Let $D(N)$ be the number of channels needed for H_r on an array of size $N = 2^r$. If a channel assignment scheme is used that is in accordance with the proof of Lemma 4, it can be shown that the equation, $D(N) = 2D(N/2) + 1$. Given that no channel is needed to realize hypercube communication on a 1–node array, D(1) $= 0$. Solving for $D(N)$ results in $D(N) = N - 1$, which is not optimal. The following lemma improves this simple channel assignment scheme.

Lemma 5: Assuming that H_{r-2} can be realized on an array of size 2^{r-2} using K channels, then H_r can be realized on an array of size 2^r using $4K+2$ channels.

Proof: Consider H_r without dimension 0 and dimension 1 connections. By the definition of H_r, $H_r - (DIM_0 \cup DIM_1) = DIM_2 \cup ... \cup DIM_{r-1}$ forms four hypercube patterns, each

```
Algorithm 1: Assign_array($N = 2^r$)
(1)  If ($r = 0$) then return $\phi$
(2)  If (r is odd) then
(3)    /* applying Lemma 4 */
(4)    recursively apply Assign_array($N/2 = 2^{r-1}$) for $EVEN_r$.
(5)    recursively apply Assign_array($N/2 = 2^{r-1}$) for $ODD_r$.
(6)    assign connections in $DIM_0$ to one channel.
(7)  Else /* r is even, apply Lemma 5 */
(8)    For i = 0, 1, 2, 3
(9)      apply Assign_array($N/4 = 2^{r-2}$) for $subarray_i$.
(10)   assign connections in $DIM_0 \cup DIM_1$ to 2 channels.
```

Figure 4.6: The channel assignment algorithm

being an H_{r-2} pattern on nodes $\{n \mid n \; mod \; 4 = i\}$ (with proper node renumbering), denoted by $subarray_i$, for $i = 0, 1, 2$ or 3. From the hypothesis, H_{r-2} can be realized on an array of size 2^{r-2} using K channels. The four sub–cube patterns can be realized in $4K$ channels. The remaining connections to be considered are those in DIM_0 and DIM_1. It can easily be proven that connections in DIM_0 and DIM_1 can be assigned to 2 channels as shown in Fig. 4.5. Hence, the hypercube communication H_r can be realized using a total of $4K + 2$ channels. □

The channel assignment algorithm, *Algorithm 1*, is depicted in Fig. 4.6. For the base case, when $N = 2^0 = 1$, the hypercube pattern contains no connection. To assign channels to connections in an array of size $N = 2^r$, $r > 0$, there are two cases. If r is even, then Lemma 5 is applied to use $4K + 2$ channels for the hypercube pattern, where K is the number of channels needed to realize a hypercube pattern on an array of size $2^{r-2} = N/4$. If r is odd, Lemma 4 is applied to use $2K + 1$ channels to realize the hypercube pattern, where K is the number of channels needed to realize a hypercube pattern in an array of size $2^{r-1} = N/2$. The example of using this algorithm to schedule H_4 in an array of size 16 is shown in Fig. 4.7.

Theorem 2: *Algorithm 1* uses $\lfloor \frac{2N}{3} \rfloor$ channels for H_r on a linear array with $N = 2^r$ nodes, thus $w(ARRAY(N), H_r) \leq \lfloor \frac{2N}{3} \rfloor$.

Proof: Let $D_{odd}(2^r)$ and $D_{even}(2^r)$ denote the number of channels needed when r is odd and even, respectively. The number of channels for the hypercube pattern using *Algorithm 1* can be formulated as follows,

$D_{odd}(2^r) = 2D_{even}(2^{r-1}) + 1$, when r is odd.

$D_{even}(2^r) = 4D_{even}(2^{r-2}) + 2$, when r is even.

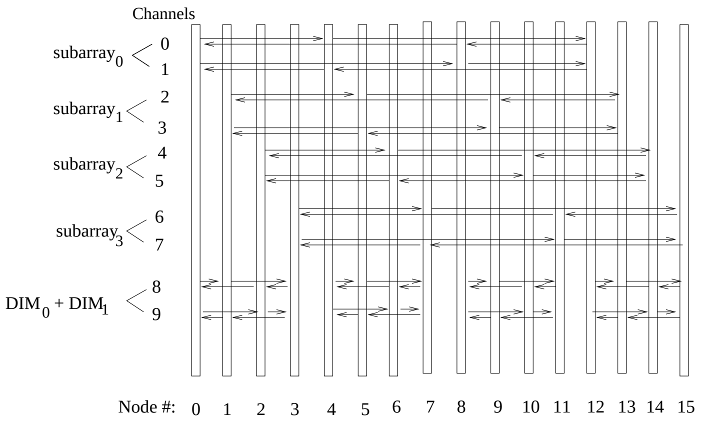


Figure 4.7: Optimal channel assignment for H_4

Using the boundary condition $D_{even}(1) = D_{even}(2^0) = 0$, it can be proven by induction that $D_{odd}(N) = \frac{2N}{3} - \frac{1}{3}$ and $D_{even}(N) = \frac{2N}{3} - \frac{2}{3}$. Hence, $D_{odd}(N)$ and $D_{even}(N)$ are equal to $\lfloor\frac{2N}{3}\rfloor$. $w(ARRAY(N), H_r) \le \lfloor\frac{2N}{3}\rfloor$. $\square$

Theorem 3:

$w(ARRAY(N), H_r) = \pi(ARRAY(N), H_r) = \lfloor\frac{2N}{3}\rfloor$, and Algorithm 1 is optimal.

Proof: Follows from Theorem 1, Theorem 2 and Lemma 1.$\square$

4.1.1.3 Hypercube connections on rings

By having links between node 0 and node $N-1$, two paths can be established from any node to any other node on a ring. It has been shown [8] that even for a fixed routing, general optimal channel assignment problem is NP–complete. This section focuses on the specific problem of optimal RCA for H_r on ring topologies, obtaining a lower bound on the number of channels needed to realize H_r and developing an optimal routing and channel assignment algorithm that achieves this lower bound.

Lemma 6: $\pi(RING(N), H_r) \ge \lfloor\frac{N}{3} + \frac{N}{4}\rfloor$.

Proof: This lemma is proven by showing that there exist two cuts on a ring that partition the ring into two sets, S_1 and S_2, such that $2 \times \lfloor\frac{N}{3} + \frac{N}{4}\rfloor$ connections in H_r originate at nodes in S_1 and terminate at nodes in S_2. Since there are only 2 links connecting S_1 to S_2, one of the 2 links must be used at least $\lfloor\frac{N}{3} + \frac{N}{4}\rfloor$ times, regardless of which routing scheme is used. Consider H_r on a ring of size $N = 2^r$. The connections in $DIM_0 \cup ...DIM_{r-2}$ form two $r-1$ dimensional hypercube patterns in two *arrays* of size 2^{r-1}. The first array, denoted by $subarray_1$, contains nodes 0, .., $2^{r-1} - 1$ and the second array, denoted by $subarray_2$, contains nodes 2^{r-1},.., $2^r - 1$. From Corollary 3.1, it follows that there exists a link in each 2^{r-1} node array such that $\lfloor\frac{N}{3}\rfloor$ connections in the hypercube pattern use that link in each direction. From the discussion in previous section, the link is $(\lfloor\frac{N}{3}\rfloor, \lceil\frac{N}{3}\rceil)$ in $subarray_1$ and

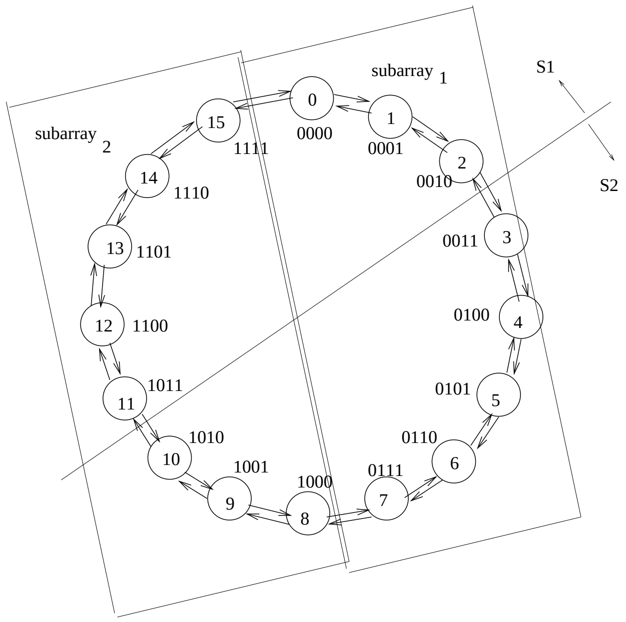


Figure 4.8: Hypercube on a ring

$(\lfloor \frac{N}{3} \rfloor + 2^{r-1}, \lceil \frac{N}{3} \rceil + 2^{r-1})$ in $subarray_2$. These two links partition the ring into two sets $S_1 = \{i | 0 \leq i \leq \lfloor \frac{N}{3} \rfloor\} \cup \{i | 2^{r-1} + \lfloor \frac{N}{3} \rfloor + 1 \leq i \leq 2^r - 1\}$ and $S_2 = \{i | \lfloor \frac{N}{3} \rfloor + 1 \leq i \leq 2^{r-1} + \lfloor \frac{N}{3} \rfloor\}$. Hence, there are $\lfloor \frac{N}{3} \rfloor$ connections from $S_1 \cap subarray_1$ to $S_2 \cap subarray_1$ and $\lfloor \frac{N}{3} \rfloor$ connections from $S_1 \cap subarray_2$ to $S_2 \cap subarray_2$ in $DIM_0 \cup ...DIM_{r-2}$. Thus, there are $2 \times \lfloor \frac{N}{3} \rfloor$ connections in $DIM_0 \cup .. \cup DIM_{r-2}$ originating at nodes in S_1 and terminating at nodes in S_2. Fig. 4.8 shows the cuts on a 16–node ring. The remaining connections of H_r are in DIM_{r-1}. By partitioning the ring into S_1 and S_2, each node in S_1 has a dimension $r-1$ connection to a node in S_2. Hence, there are $N/2$ connections in DIM_{r-1} between S_1 and S_2. Therefore, a total of $2 \times \lfloor \frac{N}{3} \rfloor + N/2 = 2 \times \lfloor \frac{N}{3} + \frac{N}{4} \rfloor$ connections in H_r are from S_1 to S_2. Thus, $\pi(RING(N), H_r) \geq \lfloor \frac{N}{3} + \frac{N}{4} \rfloor$. $\square$

The RCA scheme uses an odd–even shortest path routing. Given a ring of size $N = 2^r$, an odd–even shortest path routing works as follows. A connection between two nodes is established using a shortest path. Connections that have two shortest paths are of the forms $(i, i + 2^{r-1})$ and $(i, i - 2^{r-1})$. For these connections, the clockwise path is used if i is even and the counter–clockwise path if i is odd.

The channel assignment algorithm is derived from Lemma 6. There are two parts in the algorithm, channel assignment for connections in DIM_{r-1} and channel assignment for connections in $DIM_0 \cup .. \cup DIM_{r-2}$. Channel assignment for connections in $DIM_0 \cup .. \cup DIM_{r-2}$ is equivalent to channel assignment for two H_{r-1} in two disjoint arrays, thus, using the channel assignment scheme (for array) described in the previous section, $\lfloor \frac{N}{3} \rfloor$ channels can be used to realize these connections. For the connections in DIM_{r-1}, using odd–even

```
Algorithm 2: Assign_ring($N = 2^r$)

(1)  Apply Assign_array($N/2 = 2^{r-1}$) on $subarray_1$.
(2)  Apply Assign_array($N/2 = 2^{r-1}$) on $subarray_2$.
     Since $subarray_1$ and $subarray_2$ are disjoint,
     channels can be reused in steps (1) and (2).
(3)  for i = 0, N/2-2, step 2
       Assign a channel to connections $(i, i+2^{r-1})$, $(i+2^{r-1}, i)$,
       $(i+1, i+2^{r-1}+1)$ and $(i+2^{r-1}+1, i+1)$
```

Figure 4.9: The channel assignment for rings

shortest path routing, four connections in DIM_{r-1}, $(i, i+2^{r-1})$, $(i+2^{r-1}, i)$, $(i+1, i+2^{r-1}+1)$, $(i+2^{r-1}+1, i+1)$, can be realized using one channel. We denote by $CONFIG_i$ these four connections. Since the union of all $CONFIG_i$, where $i = 0, 2, 4, ..., N/2-2$ is equal to DIM_{r-1}, $N/4$ channels are sufficient to realize DIM_{r-1}. Fig. 4.9 shows the channel assignment algorithm for ring topologies.

Theorem 4: *Algorithm 2* uses $\lfloor \frac{N}{3} + \frac{N}{4} \rfloor$ channels to realize H_r in a ring of size $N = 2^r$.

Proof: Follows from above discussion. □

Theorem 5: $w(RING(N), H_r) = \pi(RING(N), H_r) = \lfloor \frac{N}{3} + \frac{N}{4} \rfloor$, and the odd–even shortest path routing with Algorithm 2 is an optimal RCA scheme for hypercube connection on rings.

Proof: Follows from Lemma 1, Lemma 6 and Theorem 4.□

4.1.1.4 Hypercube connections on meshes

Given a $2^k \times 2^{r-k}$ mesh, realizing the hypercube connections on the mesh is equivalent to realizing H_k in each row and H_{r-k} in each column. The following lemma gives the lower bound on the number of channels required to realize hypercube communication patterns on meshes.

Lemma 7: $\pi(MESH(2^k \times 2^r - k), H_r) \geq \lfloor \frac{2 \times 2^k}{3} \rfloor$, assuming $k \geq r - k$.

Proof: The hypercube pattern on the mesh contains 2^{r-k} k–dimensional hypercube patterns on 2^k arrays in the 2^{r-k} rows. Consider a cut in edges $(\lfloor \frac{2^k}{3} \rfloor, \lceil \frac{2^k}{3} \rceil)$ in every row, which partitions the mesh into two parts. From Corollary 3.1, we know that for each row there are $\lfloor \frac{2 \times 2^k}{3} \rfloor$ connections from the left of the cut to the right of the cut, hence, there are a total of $2^{r-k} \times \lfloor \frac{2 \times 2^k}{3} \rfloor$ connections crossing the cut. Since there are 2^{r-k} edges in the cut, there exists at least one edge that is used at least $\lfloor \frac{2 \times 2^k}{3} \rfloor$ times. Thus, $\pi(MESH(2^k \times 2^r - k), H_r) \geq \lfloor \frac{2 \times 2^k}{3} \rfloor$. □

Given a mesh of size $2^k \times 2^{r-k}$, the hypercube communication pattern in each row is denoted by H_k^{row} and the hypercube communication pattern in each column by H_{r-k}^{col}. The

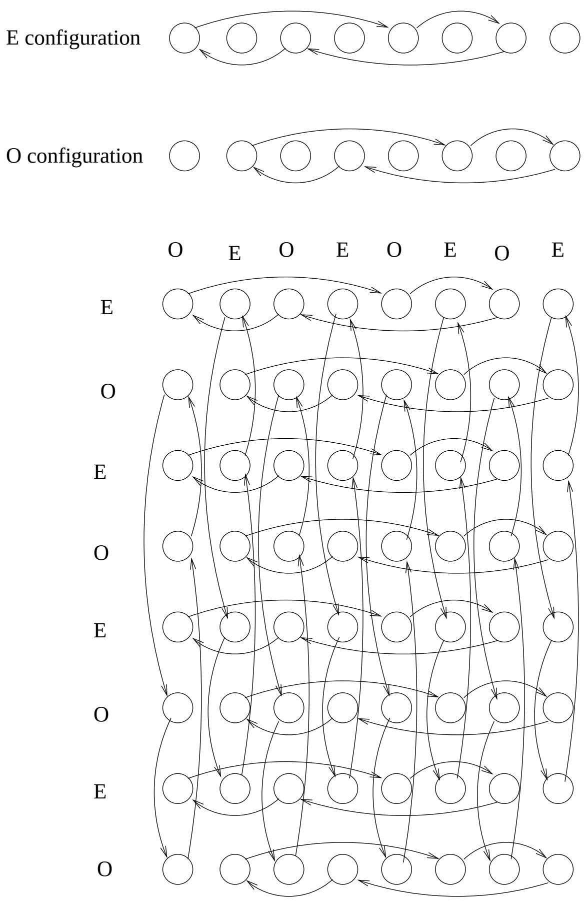


Figure 4.10: a Mesh configuration

RCA scheme uses X–Y shortest path routing. Since we already know the optimal channel assignment for H_k^{row} and H_{r-k}^{col}, the challenge here is to reuse channels on connections in two dimensions efficiently. Let us define an *array configuration* as the set of connections in a linear array that are assigned to the same channel. *Ring*, *mesh* and *torus configurations* are defined similarly. Using the definition of configurations, a mesh configuration can be obtained by combining array configurations in the rows and the columns. For example, if an array configuration in x dimension and an array configuration in y dimension can be combined into a mesh configuration, the two array configurations can be realized in the mesh topology using one channel. Notice that, while there is no link conflict when assigning channels to row and column connections, node conflicts may occur and must be avoided.

Let us first take a deeper look at the array configurations for arrays of size $N = 2^k$. Following the channel assignment algorithm, *Algorithm 1*, array configurations can be classified into three categories; E–configurations that contain only connections between even–numbered nodes, O–configurations that contain only connections between odd–numbered nodes, and EO–configurations that contain dimension 0 (and/or) dimension 1 connections

As discussed in Section 3, if k is odd, there is only one EO–configuration for connections in DIM_0, $(\lfloor\frac{2N}{3}\rfloor - 1)/2$ E–configurations for connections in $EVEN_k$, and $(\lfloor\frac{2N}{3}\rfloor - 1)/2$ O–configurations for connections in ODD_k. Similarly, if k is even, there are two EO–configurations, $(\lfloor\frac{2N}{3}\rfloor - 2)/2$ E–configurations and $(\lfloor\frac{2N}{3}\rfloor - 2)/2$ O–configurations. The following lemma shows that E–configurations and O–configurations in rows and columns of the mesh can be combined.

Lemma 8: Given an E–configuration, E_x, and an O–configuration, O_x, in the x direction and an E–configuration, E_y, and an O–configuration, O_y, in the y direction, E_x and O_x in all rows and E_y and O_y in all columns can be realized in two mesh configurations.

Proof: The proof is by constructing the two mesh configurations. In the first mesh configuration, let all odd numbered rows realize O_x and all even numbered row realize E_x. In this case, no connection starts or terminates at an odd numbered node in an even column or at an even numbered node in an odd column. Thus, in the same mesh configuration, E_y can be realized in odd columns and O_y can be realized in even columns. The second mesh configuration realizes E_x on odd numbered rows, O_x on even numbered rows, E_y on even numbered columns and O_y on odd numbered columns. These two mesh configurations realize E_x and O_x in all rows and E_y and O_y in all columns. Fig. 4.10 shows the construction of a mesh configuration. $\square$

Lemma 8 lays the foundation for the channel assignment algorithm. Let a be the number of E–configurations and O–configurations in H_k^{row}, b be the number of EO–configurations in H_k^{row}, c be the number of E–configurations and O–configurations in H_{r-k}^{col}, and d be the number of EO–configurations in H_{r-k}^{col}. From assumptions, it follows that $k \geq r-k$, $a \geq c$, $a+b = \lfloor\frac{2\times 2^k}{3}\rfloor$ and $d \leq 2$. By combining E–configurations and O–configurations in rows and columns into mesh configurations, all the E–configurations and O–configurations in each row and all the E–configurations and O–configurations in each column can be realized using a mesh configurations. Using an individual mesh configuration for each EO configuration in the rows and the columns, a total of $a+b+d \leq \lfloor\frac{2\times 2^k}{3}\rfloor + 2$ configurations are sufficient to realize the hypercube connections.

Theorem 4: H_r can be realized on a $2^k \times 2^{r-k}$ mesh, where $k \geq r-k$, using $\lfloor\frac{2\times 2^k}{3}\rfloor + 2$ channels. $\square$

Corollary 4.1: $w(MESH(2^k \times 2^{r-k}), H_r) \leq \lfloor\frac{2\times 2^k}{3}\rfloor + 2 \leq \pi(MESH(2^k \times 2^{r-k}), H_r) + 2$. $\square$

4.1.1.5 Hypercube connections on tori

As in the case of realizing H_r on a mesh, H_r can be realized on a $2^k \times 2^{r-k}$ torus by realizing H_k^{row} in each row and H_{r-k}^{col} in each column. The following lemma gives a lower bound on the number of channels required to realize H_r on a torus.

Lemma 9: $\pi(TORUS(2^k \times 2^r - k), H_r) \geq \lfloor \frac{2^k}{3} + \frac{2^k}{4} \rfloor$, assuming $k \geq r - k$.

Proof: The hypercube pattern on the torus contains 2^{r-k} k–dimensional hypercube patterns on 2^k rings in the 2^k rows. Considered two cuts in edges $(\lfloor \frac{2^{k-1}}{3} \rfloor, \lceil \frac{2^{k-1}}{3} \rceil)$ and $(\lfloor \frac{2^{k-1}}{3} \rfloor + 2^{k-1}, \lceil \frac{2^{k-1}}{3} \rceil + 2^{k-1})$ in every row which partition the torus into two parts. Following the same reasoning as in the proof of lemma 6, it is known that for each row there are $2 \times \lfloor \frac{2^k}{3} + \frac{2^k}{4} \rfloor$ connections from one part to the other part, hence, there are a total of $2^{r-k} \times 2 \times \lfloor \frac{2^k}{3} + \frac{2^k}{4} \rfloor$ connections crossing the two parts. Since there are $2 \times 2^{r-k}$ edges in the cut, regardless of the routing scheme used, there exist at least one edge that is used at least $\lfloor \frac{2^k}{3} + \frac{2^k}{4} \rfloor$ times. Thus, $\pi(TORUS(2^k \times 2^r - k), H_r) \geq \lfloor \frac{2^k}{3} + \frac{2^k}{4} \rfloor$. $\Box$

X–Y routing between dimensions and odd–even shortest path routing within each dimension are used to develop the RCA scheme. Next, the combination of ring configurations into torus configurations is considered. As in the case of rings, given a $2^k \times 2^{r-k}$ torus, the connections in H_r are partitioned into two sets. The first set includes all connections in $DIM_0 \cup .. \cup DIM_{k-2}$ in each row and all connections in $DIM_0 \cup .. \cup DIM_{r-k-2}$ in each column. The second set includes the connections in DIM_{k-1} in each row and the connections in DIM_{r-k-1} in each column. The connections in $DIM_0 \cup .. \cup DIM_{k-2}$ in each row and the connections in $DIM_0 \cup .. \cup DIM_{r-k-2}$ in each column form four hypercube patterns on four disjoint $2^{k-1} \times 2^{r-k-1}$ sub–meshes in the torus. A straight forward extension of the channel assignment scheme in the previous section can be used to assign channels to these connections with at most $\lfloor \frac{2^k}{3} \rfloor + 2$ channels.

To realize the connections in DIM_{k-1} in each row and the connections in DIM_{r-k-1} in each column, The same partitioning for the ring topology discussed in section 4 is followed. Specifically, the following configurations are constructed in rows and columns respectively

$row_i = \{(i, i + 2^{k-1}), (i + 2^{k-1}, i), (i + 1, i + 1 + 2^{k-1}), (i + 1 + 2^{k-1}, i + 1)\}$

$column_j = \{(j, j + 2^{r-k-1}), (j + 2^{r-k-1}, j), (j + 1, j + 1 + 2^{r-k-1}), (j + 1 + 2^{r-k-1}, j + 1)\}$

DIM_{k-1} is composed of the configurations row_i, for $i = 0, 2, ..., 2^{k-1} - 2$ and DIM_{r-k-1} is composed of the configurations $column_j$ for $j = 0, 2, ..., 2^{r-k-1} - 2$.

Lemma 10 For any i_1, i_2, where $i_1 \neq i_2$, row_{i_1} and row_{i_2} in each row and $column_{i_1}$ and $column_{i_2}$ in each column can be realized in two torus configurations.

Proof: Similar to the proof of Lemma 8, omitted. $\Box$

Theorem 5: H_r can be realized on a $2^k \times 2^{r-k}$ torus, where $k \geq r-k$, using $\lfloor \frac{2^k}{3} + \frac{2^k}{4} \rfloor + 2$ channels.

Proof: As discussed above, $\lfloor \frac{2^k}{3} \rfloor + 2$ channels are sufficient to realize all connections in H_r, except the connections in DIM_{k-1} in each row and DIM_{r-k-1} in each column, by realizing four hypercube communication patterns on the four disjoint sub–meshes. From Lemma 10, configurations row_i, $i = 0, 2, ..., 2^{r-k-1} - 2$ and configurations $column_j$, $j = 0, 2, ..., 2^{r-k-1} - 2$ can be realized in 2^{r-k-2} torus configurations. Since $2^{k-2} - 2^{r-k-2}$ torus configurations can be used to realize row_i, $i = 2^{r-k-1}, 2^{r-k-1} + 2, .., 2^{k-1} - 2$, all the dimension $k-1$ connections in each row and dimension $r-k-1$ connections in each column can be realized in 2^{k-2} torus configurations. Hence, H_r can be realized by a total of $\lfloor \frac{2^k}{3} \rfloor + 2 + 2^{k-2} = \lfloor \frac{2^k}{3} + \frac{2^k}{4} \rfloor + 2$ configurations. □

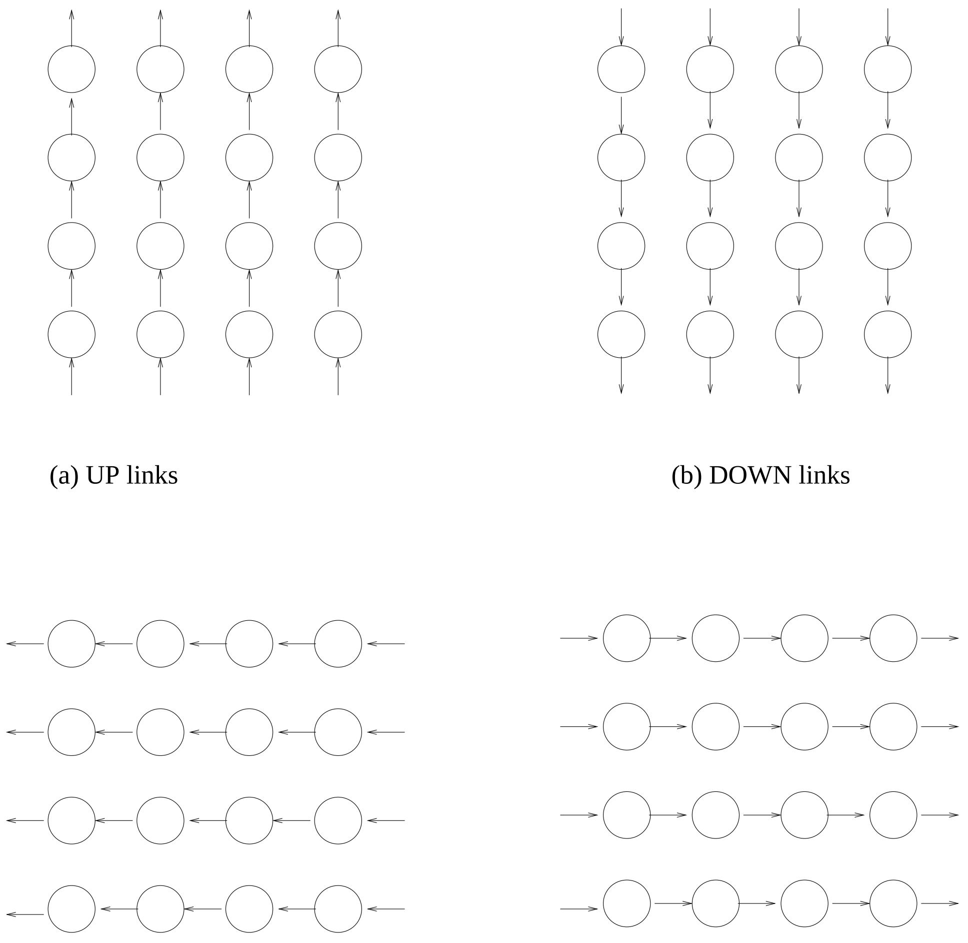
Corollary 5.1: $w(TORUS(2^k \times 2^{r-k}), H_r) \leq \lfloor \frac{2^k}{3} + \frac{2^k}{4} \rfloor + 2 \leq \pi(TORUS(2^k \times 2^{r-k}), H_r) + 2$. □

4.1.2 Logical torus, all–to–all and allXY topologies

The logical torus topology coincides with the physical network. Thus, when realizing logical torus topology, there are no link conflicts since the physical network can support all links in the logical network simultaneously. However, node conflicts may occur. Under our network model, each node in the network can only access one channel at any given time slot. Hence, to support 4 out–going links at each node, at least 4 channels are needed. Using 4 channels, the logical torus topology can be realized as follows. All links in a torus can be classified into four categories, the UP links, the DOWN links, the LEFT links and the RIGHT links. Each category can be realized using 1 channels without incurring node conflicts and link conflicts as shown in Figure 4.11. Notice that all nodes can be sending and receiving messages in the figure. Hence, 4 channels are sufficient and necessary to realize the logical torus topology on top of the physical torus topology.

Optimal schemes to realize all–to–all communication on ring and torus topologies can be found in [33]. It is shown in [33] that for an N node ring, $N \geq 8$, the all–to–all communication can be realized with $N^2/8$ channels without node conflicts. For an $N \times N$ torus, the all–to–all communication can be realized with $N^3/8$ channels. The connections on each channel to realize the all–to–all communication will be called an *AAPC configuration.* Details about the connection scheduling can be found in [33].

The logical allXY topology realizes all–to–all connections in each dimension in the physical torus. For an $N \times N$ torus, each node in the logical allXY topology logically connects to $2N-2$ nodes. Using the AAPC configurations for rings, techniques similar to



(a) UP links

(b) DOWN links

(c) LEFT links

(d) RIGHT links

Figure 4.11: Realizing logical torus topology

the ones in section 4.1.1.5 can be used to combine the ring configurations to form torus configurations and realize the allXY on an $N \times N$ torus, where $N \geq 16$, resulting in a multiplexing degree of $N^2/8$. For an $N \times N$ torus with $N \leq 8$, $2N-2$ channels can be used to realize the allXY topology. For example, using the 8 AAPC configurations for 8–node rings in [33], 6 configurations along each dimension cannot be combined because of node conflicts, while 2 configurations in each dimension can be combined in the torus, resulting a multiplexing degree of $14 = 2 \times 8 - 2$ for realizing the allXY topology.

4.2 Performance of the logical topologies under light load

This section considers the communication performance of the logical topologies under light load such that the network contentions on both channels and switches are negligible. An analytical model will be described that takes the network contention effect into consideration later in this chapter.

Let us assume that a packet can be transferred from source to destination on a path in one time slot and that the network has a multiplexing degree of d. It takes on average $\frac{d+1}{2}$ time slots to transfer a packet from a router to the next router. Thus, assuming that the packet routing time in each router (including the E/O, O/E conversions) is γ, the average number of intermediate routing hops per packet is h, and the network contention is negligible, the average delay time for each packet can be expressed as follows:

$$delay = (h+2) * \gamma + (h+1) * \frac{d+1}{2}.$$

The first term, $(h+2) * \gamma$, is the average routing time that a packet spends at the h intermediate routers and the 2 routers at the sending and receiving nodes. The second term, $(h+1) * \frac{d+1}{2}$, is the average packet transmission time on paths plus the time that a packet waits in the output path buffers. Thus, the average delay time is determined by three parameters, the multiplexing degree d, the packet routing time γ, and the average number of hops per packet transmission h. We can assume that the packet routing time γ is the same for all topologies. Different logical topologies result in different number of intermediate hops, h, and different multiplexing degree, d. Next, the performance of the four logical topologies is discussed.

Given an $N \times N$ torus, the logical all–to–all topology establishes direct connections between all pairs of nodes and thus, totally eliminates the intermediate hops, resulting in $h = 0$. Using the algorithm in [33], a multiplexing degree of $\frac{N^3}{8}$ can be used to realize the logical all–to–all topology. Thus $d = \frac{N^3}{8}$, and the delay time is given by:

$$delay_{all-to-all} = 2 \times \gamma + (\frac{N^3}{8} + 1)/2 = O(\gamma + N^3).$$

Given an $N \times N$ torus, a logical torus topology can be realized using a multiplexing degree of 4 (i.e., $d = 4$). For a logical $N \times N$ topology, the average number of intermediate hops is $h = \frac{N}{2} - 1$. Hence the delay time for the logical torus topology is given by:

$$delay_{torus} = (\frac{N}{2} + 1) \times \gamma + \frac{N}{2} \times (4 + 1)/2 = O(N \times \gamma).$$

For $N = 2^r$, the algorithm in section 4.1.1.5 can realize a logical hypercube topology on an $N \times N$ torus using a multiplexing degree of $\lfloor \frac{N}{3} + \frac{N}{4} \rfloor + 2$, if r is odd, and $\lfloor \frac{N}{3} + \frac{N}{4} \rfloor + 1$, if r is even. For a logical N^2 node hypercube, the average number of intermediate hops is $h = \frac{lg(N^2)}{2} - 1 = lg(N) - 1$. Hence, the delay time (for an even r) is given by:

$$delay_{hypercube} = (lg(N) + 1) \times \gamma + \lg(N) \times (\lfloor \frac{N}{3} + \frac{N}{4} \rfloor + 2)/2 = O(\gamma lg(N) + N lg(N)).$$

Finally, let us consider the logical *allXY* topology. As discussed in section 4.1.2, when $N \leq 8$, the logical topology can be realized using a multiplexing degree of $2N - 2$. For $N > 8$, a multiplexing degree of $\frac{N^2}{8}$ is needed. Since for two nodes in the same column or row, no intermediate hop is needed, while in other cases, one intermediate hop is required, the average number of intermediate hops on the logical allXY topology is given by:

$$\frac{2N-2}{N^2-1} \times 0 + \frac{(N^2-1)-(2N-2)}{N^2-1} \times 1 = \frac{N^2-2N+1}{N^2-1}.$$

Therefore, for $N > 8$, the average delay can be expressed as follows:

$$delay_{all\_XY} = (2 + \frac{N^2 - 2N + 1}{N^2 - 1}) \times \gamma + (1 + \frac{N^2 - 2N + 1}{N^2 - 1}) \times (\frac{N^2}{8} + 1)/2 = O(\gamma + N^2).$$

Logical topology	Number of intermediate hops (h)	multiplexing degree (d)	total number of connections (P)
all–to–all	0	$\frac{N^3}{8}$	$N^2(N^2 - 1)$
all_XY	$\frac{N^2-2N+1}{N^2-1}$	$\frac{N^2}{8}$ †	$N^2(2N - 2)$
hypercube	$lg(N) - 1$	$\lfloor \frac{N}{3} + \frac{N}{4} \rfloor + 1$ ‡	$N^2 lg(N)$
torus	$\frac{N}{2} - 1$	4	$N^2 \times 4$

† Assuming that $N > 8$. If $N < 8$, the value is $2N - 2$.
‡ Assuming that r is even. If r is odd, the value is $\lfloor \frac{N}{3} + \frac{N}{4} \rfloor + 2$.

Table 4.1: Summary of logical topologies

Table 4.1 summarizes the average number of intermediate hops (h), the multiplexing degree (d) and the total number of logical connections (P) for the four topologies. Figure 4.12 plots the average delay as a function of the packet routing time γ, for the four logical topologies on a physical 16×16 torus. When γ is very small compared to data transmission time ($\gamma \leq 0.5$), the logical torus topology achieves the smallest delay time. When $0.5 \leq \gamma \leq 4.25$, the logical hypercube has the best performance. When $4.25 \leq \gamma \leq 256.25$, the allXY topology gives the best performance. When $\gamma > 256.25$, the all–to–all topology has the smallest packet delay.

The characteristics exhibited in Figure 4.12 are true for any network size. Specifically, for a given N, there is a value of γ below which routing on the torus is more efficient than routing on the logical hypercube. Similarly, there is a value of γ, above which routing on the allXY topology is more efficient than routing on the logical hypercube. Finally, there is a value of γ, above which routing on the all–to–all topology is more efficient than routing on the allXY topology. In Figure 4.13 these special values are plotted for different N and the (N, γ) parameter space is divided into four regions. Each region is labeled by the logical topology that results in the lowest average packet delay.

These results are obtained by ignoring network traffic contention, and thus are valid only under light load. In the next section, a queuing model is used to study the network performance under high load.

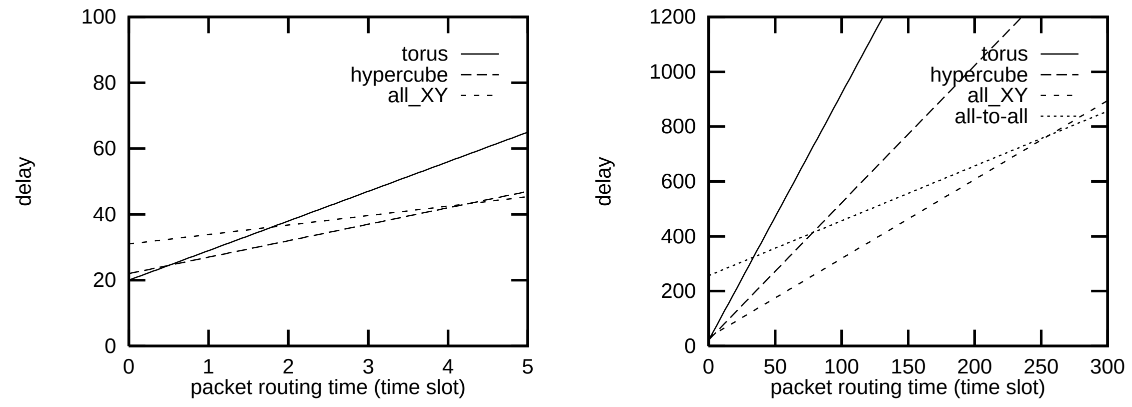


Figure 4.12: Performance for logical topologies on 16×16 torus

4.3 An analytical model and its verification

This section describes an approximate analytical model that takes network contention into consideration. This model is used to study the effect of the network load on the maximum throughput and the packet delay. It is assumed that in each time slot, a

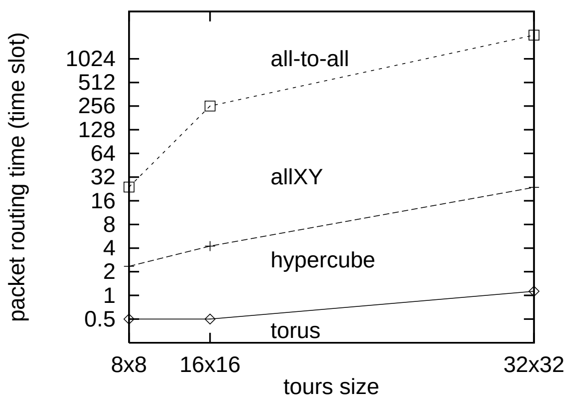


Figure 4.13: Logical topologies giving lowest packet delay for given γ and N

packet can be sent from the source to the destination on a path. For example, if a 1Gbps channel is used with a 53–byte packet (or cell) as defined in the ATM standard, then the slot duration is $0.424\mu s$. All other delays in the system are normalized with respect to this slot duration.

The routers and the paths in a network are modeled as a network of queues. As shown in Figure 4.1, each router has a routing queue that buffers the packets to be processed. The router places packets either into one of the output path queues that buffer packets waiting to be transmitted, or into the local processor. Both a router and a path have a constant service time. The exact model for such network is very difficult to obtain. The network is approximated by making the following assumptions: 1) each queue is independent of each other and 2) each queue has a Poisson arrival and constant service time. These assumptions enable the derivation of expressions for the maximum throughput and the average packet delay of the four logical topologies by dealing with the M/D/1 queues independently. The simulation results confirm that these approximations are reasonable. The following notation is used in the model:

- N. Size of each dimension of the torus. Thus, the network has a total of N^2 nodes.
- d, h and P are defined in the previous section. A *frame* consists of d time slots. Within a frame, one time slot is allocated to each path. As discussed earlier, the average number of paths that a packet traverses is equal to $h + 1$. The average number of routers that a packet traverses is $h + 2$.
- λ. Average packet generation rate at each node per time slot. This implies that the average generation rate of packets to the entire network is $N^2\lambda$. It is assumed that the arrival process is Poisson and is independently and identically distributed on all network nodes. Furthermore, it is assumed that all packets are equally likely to be

destined to any one of the network nodes. At each router, the newly generated packets and the packets arriving from other nodes are maintained in an infinite routing buffer before being processed as shown in Figure 4.1.

- λ_s. Average rate of packet arrival at a router per time slot, including both generated packets and packets received from other nodes. This composite arrival rate, λ_s, may be derived as follows. In any time slot the total number of generated packets that arrive at all the routing buffers is λN^2. On average, each of these packets traverses $h+2$ routers within the network. Therefore, under steady state condition, there will be $\lambda N^2(h+2)$ packets in all the routers of the network in each time slot. Under the assumption that each packet is equally likely to be in each router, the total arrival rate is given by $\lambda_s = \lambda(h+2)$.

- λ_p. Average rate of packet arrival at a path buffer per time slot. This arrival rate, λ_p, can be derived as follows. Under steady state condition, in any time slot, the total number of packets in all the routers in the network is $\lambda N^2(h+2)$. Of all these packets, λN^2 packet will exit the network and $\lambda N^2(h+2) - \lambda N^2 = \lambda N^2(h+1)$ packets will be transmitted through paths in the network. Under the assumption that sources and destinations are uniformly distributed in the network, the average arrival rate is given by $\lambda_p = \frac{\lambda N^2(h+1)}{P}$.

- γ. The routing time per packet at a router. Since packets are of the same length, the routing time is a constant value. The average packet departure rate from the routing buffer, denoted by μ_s, is $\mu_s = \frac{1}{\gamma}$.

- μ_p. The average packet departure rate from each path buffer per time slot. Since in the model used, each path will be served once in every frame, $\mu_p = \frac{1}{d}$. The average service time in each path is $S_p = \frac{1}{\mu_p} = d$.

Maximum throughput

With the above notation, the maximum throughput and average packet delay of the logical topologies can now be studied. First the theoretical maximum throughput is considered and then the average packet delay. Two bottlenecks can potentially limit the maximum throughput.

- If the average packet arrival rate at a routing buffer is larger than the average packet departure rate, that is if $\lambda_s \leq \mu_s$, then the throughput will be limited by the router processing bandwidth. The maximum packet generation rate allowed by the router bandwidth, λ_s^{max}, can be derived as follows: $\lambda_s \leq \mu_s$, or $(h+2)\lambda \leq \frac{1}{\gamma}$, or $\lambda \leq \frac{1}{\gamma(h+2)}$. Thus,

$$\lambda_s^{max} = \frac{1}{\gamma(h+2)}$$

- If the average packet arrival rate at a path buffer is larger than the average packet departure rate, that is $\lambda_p \leq \mu_p$, then the throughput will be limited by the path bandwidth. The maximum fresh packet generation rate allowed by the path bandwidth, λ_p^{max}, can be derived as follows: $\lambda_p \leq \mu_p$, or $\frac{(h+1)\lambda N}{P} \leq \frac{1}{d}$, or $\lambda \leq \frac{P}{(h+1)Nd}$. Thus,

$$\lambda_p^{max} = \frac{P}{(h+1)Nd}$$

The theoretical maximum throughput is the minimum of λ_s^{max} and λ_p^{max}, that is, $\lambda^{max} = min(\lambda_s^{max}, \lambda_p^{max})$. Given a topology, $\lambda^{max} = \lambda_s^{max}$ indicates that the router speed is the bottleneck, while $\lambda^{max} = \lambda_p^{max}$ indicates that the path speed is the bottleneck.

Average packet delay

As mentioned before, the packet delay is divided into the *routing delay*, which includes the time a packet spends on routing buffers and the time for routers to process the packets, and the *transmission delay*, which includes the time a packet spends on path buffers and the actual packet transmission time on the paths.

Let us first consider the routing delay in each router. It takes γ timeslots for a router to process the packet when the packet reaches the front of the routing buffer. As for the packet waiting time in the routing buffer, since the routing buffer is modeled as an $M/D/1$ queue, the average queuing delay depends on the arrival rate λ_s and is given by:

$$Q = \frac{\lambda_s(\gamma)^2}{2(1-\frac{\lambda_s}{\mu_s})}$$

where λ_s is the average packet arrival rate, γ is the expected service time, μ_s is the average packet departure rate. Given that $\mu_s = \frac{1}{\gamma}$, the total time that a packet spends in each router is given by:

$$routing\ delay = \gamma + \frac{\lambda_s(\gamma)^2}{2(1-\lambda_s\gamma)} \qquad (1)$$

Consider the two components of the transmission delay on each path. The first component is the delay required by a packet to synchronize with the appropriate outgoing slot in the frame on which the node transmits and the actual packet transmission time. The average value of this delay is $\frac{1+2+...+d}{d} = \frac{d+1}{2}$. The second component is the $M/D/1$ queuing delay that a packet experiences at the buffer before it reaches the head of the buffer. This follows the same formula as in the routing delay case, and is given by,

$$\frac{\lambda_p S_p^2}{2(1-\frac{\lambda_p}{\mu_p})} = \frac{\lambda_p d^2}{2(1-\lambda_p d)}$$

The two components are combined to obtain the total delay a packet encounters on a path as follows,

$$transmission\ delay = \frac{d+1}{2} + \frac{\lambda_p d^2}{2(1-\lambda_p d)} \qquad (2)$$

As discussed earlier, each packet takes $h+2$ hops and $h+1$ paths on average. Thus, given that on average, a packet spends *routing delay* in each router and *transmission delay* on each path, the average packet delay can be expressed as follows:

$$delay = (h+2) \times routing\ delay + (h+1) \times transmission\ delay.$$

Using formula (1) and (2), the following average delay encountered by a packet from the source to the destination is obtained.

$$delay = (h+2) \times (\gamma + \frac{\lambda_s(\gamma)^2}{2(1-\lambda_s\gamma)}) + (h+1) \times (\frac{d+1}{2} + \frac{\lambda_p d^2}{2(1-\lambda_p d)})$$

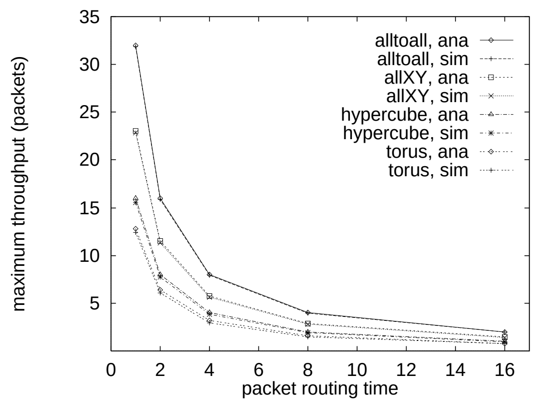
Model verification

To verify the analytical model and to further study the performance of these logical topologies, a network simulator was developed that simulates all four logical topologies on top of the torus topology. The simulator takes the following parameters.

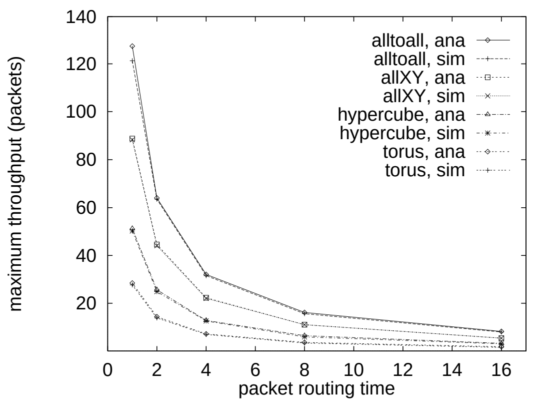
- *system size*, $N \times N$: This specifies the size of the network. Based on the logical topology, the system size also determines the multiplexing degree in the system.
- *packet generation rate*, λ: This is the rate at which fresh packets are generated in each node. It specifies the traffic on the network. The inter–arrival of packets follows a Poisson distribution. When a packet is generated at a node, the destination is generated randomly among all other nodes in the system with a uniform distribution.
- *Packet routing time*, γ.

Fig 4.14 shows the maximum throughput obtained from the analytical model and from simulations. Both 8×8 and 16×16 physical torus networks with different packet routing time are examined. As can be seen from the figure, the analytical results and the simulation results almost have a perfect match for all cases.

Figure 4.15 and Figure 4.16 show the average packet delays obtained from the analytical model and from simulations. Here, the packet routing time, γ, is equal to 1



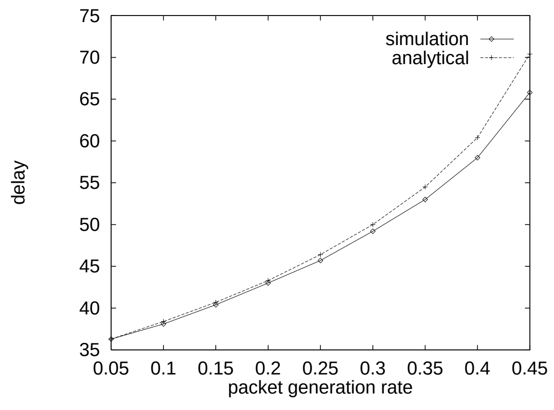
(a) physical 8 × 8 torus



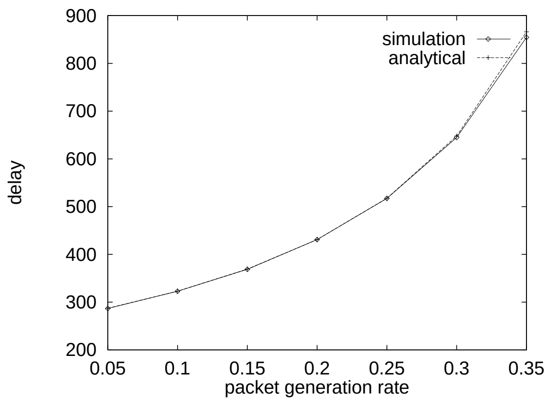
(b) physical 16 × 16 torus

Figure 4.14: predicted and simulated maximum throughput

time slot. For 8 × 8 torus, the analytical model matches the simulation results fairly well for all topologies except when the generation rate is close to saturation. The difference between the results from the analytical model and those from simulations is around 10%. For the 16 × 16 physical topology, the analytical model matches the simulations results for the all–to–all, allXY and hypercube topologies. For the torus topology, the difference is about 20% due to the approximation. Studies using other values of γ have also been conducted. The analytical model and the simulation results on those studies match slightly better than those shown in Figures 4.15 and 4.16. Thus, overall the analytical model gives a good indication of the actual performance.

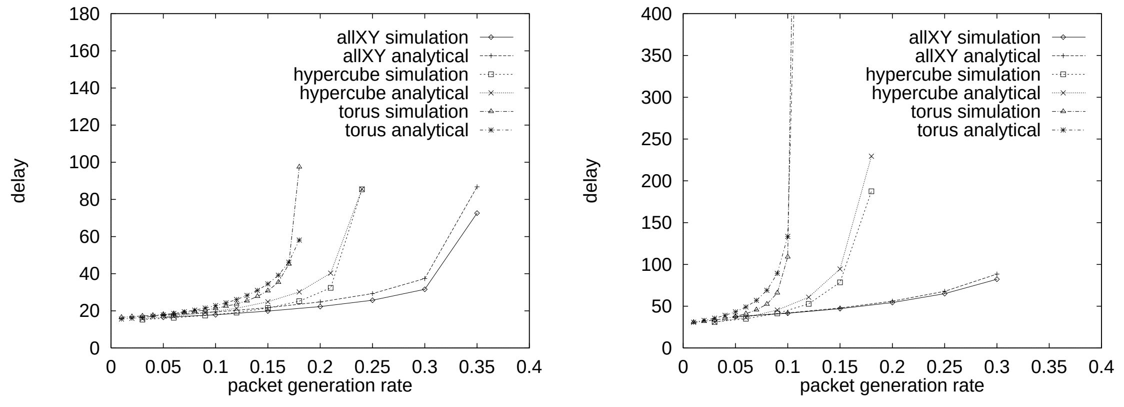


(a) physical 8 × 8 torus



(b) physical 16 × 16 torus

Figure 4.15: Packet delays for logical all–to–all topology ($\gamma = 1$)



(a) physical 8×8 torus (b) physical 16×16 torus

Figure 4.16: Packet delays for logical allXY, hypercube and torus topologies ($\gamma = 1$)

4.4 Performance of the logical topologies

In the previous section, an analytical model for performance study for the logical topologies was developed and verified. This section focuses on studying the performance of the logical topologies. Since the simulation and the analytical model match reasonably well, only the analytical model is used in this section to study the performance.

Figure 4.17 shows the impact of packet routing time on the maximum throughput. The underlying topology is a 32×32 torus. For all logical topologies, increasing the speed of routers increases the maximum throughput up to a certain limit. For the all–to–all topology, the router speed of 1 packet per 4 time slots is sufficient to overcome the router performance bottleneck. Using faster router will not further improve the maximum throughput. For the allXY and hypercube topologies, the threshold is 1 packet per 2 time slots, and for the torus topology, the threshold is 1 packet per time slot. When the routing speed is faster than the threshold value, the maximum throughput is bound by the link speed and the maximum throughput will not increase along with the increase in router speed. Table 4.2 shows the bandwidth limits of routers and links for $N = 32$.

Figure 4.17 also shows that the all–to–all topology achieves higher maximum throughput than the allXY topology, which in turn achieves higher maximum throughput than the hypercube topology. The logical torus has the worst maximum throughput. This observation holds for all packet routing speeds. Under high workload, all paths in the all–to–all and allXY topologies are utilized. The algorithms to realize the all–to–all and allXY topologies guarantee that in each time slot all links are used if all connections scheduled for that time slot are in use, while the hypercube and torus topologies can not achieve this effect. Thus, it is expected that the all–to–all topology and the allXY topology will outperform the hypercube and torus topologies in terms of maximum throughput.

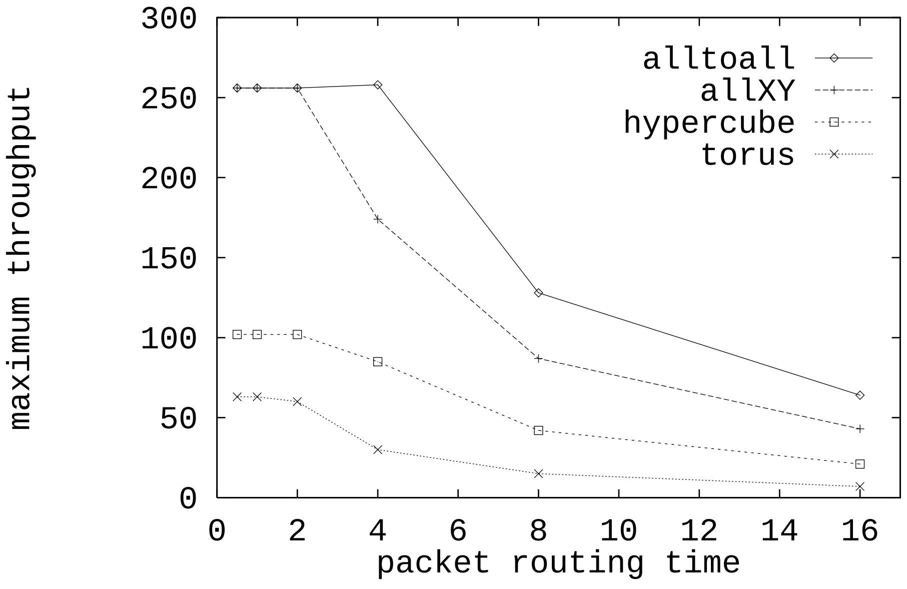


Figure 4.17: Maximum throughput .vs. packet routing time ($N = 32$)

Figure 4.18 shows the impact of network size on the maximum throughput. The results in this figure are based upon a packet routing time of one time slot. Different packet routing times were also studied and similar trends were found. In terms of maximum throughput, the all–to–all topology scales the best, followed by the allXY topology, followed by the hypercube topology. The logical torus topology scales worst among all these topologies. Figures 4.17 and 4.18 show that by using time–division multiplexing to establish complex logical topology, the large aggregate bandwidth in the network can be exploited to deliver higher throughput when the network is under high workload.

Although the all–to–all topology is the best in terms of the maximum throughput, it suffers from large packet delay when the network is not saturated. Packet delay is another performance metric to be considered. For a network to be efficient, it must also be able to deliver packets with a small delay. It is well known that time–division multiplexing results in larger average packet delay due to the sharing of the links. However, as discussed earlier, while using time–division multiplexing techniques to establish logical topologies increases the per hop transmission time, it reduces the average number of hops that a packet travels. Thus, the overall performance depends on system parameters. Next, this effect for the logical topologies is studied.

topology	bottleneck	$\gamma = 0.5$	$\gamma = 1$	$\gamma = 2$	$\gamma = 4$
all–to–all	λ_s^{max}	2.0	1.0	0.5	0.25
	λ_p^{max}	0.25	0.25	0.25	0.25
	λ^{max}	0.25	0.25	0.25	0.25
allXY	λ_s^{max}	1.36	0.68	0.34	0.17
	λ_p^{max}	0.25	0.25	0.25	0.25
	λ^{max}	0.25	0.25	0.25	0.17
hypercube	λ_s^{max}	0.67	0.33	0.17	0.09
	λ_p^{max}	0.1	0.1	0.1	0.1
	λ^{max}	0.1	0.1	0.1	0.09
torus	λ_s^{max}	0.24	0.12	0.06	0.03
	λ_p^{max}	0.06	0.06	0.06	0.06
	λ^{max}	0.06	0.06	0.06	0.03

Table 4.2: Maximum throughput for the logical topologies on 32×32 torus

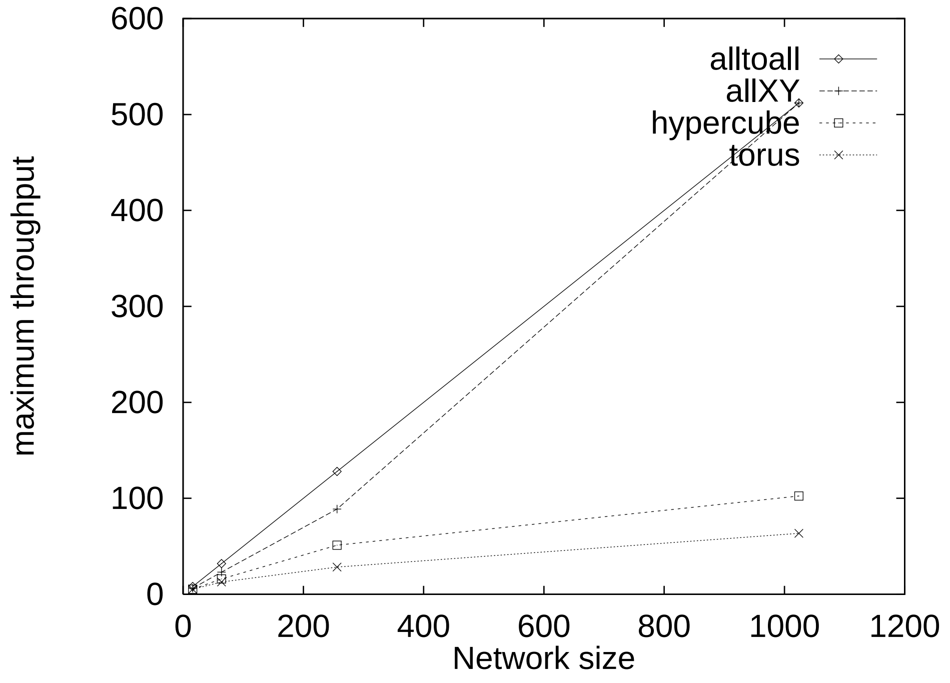


Figure 4.18: Maximum throughput .vs. network size ($\gamma = 1$)

Figure 4.19 shows the delay with regard to the fresh packet generation rate. The underlying topology is a 16×16 torus and γ is 1 time slot. The figure shows that the all–to–all topology incurs very large delay compared to other logical topologies. This is because of the large multiplexing degree needed to realize the logical all–to–all topology. Other topologies have similar delay when the generation rate is small, that is, under low workload. However, the allXY topology has a larger saturation point than the hypercube and torus topologies and thus has a small delay even when the network load is reasonably high (e.g. $\lambda = 0.25$). These results also hold for larger packet routing times.

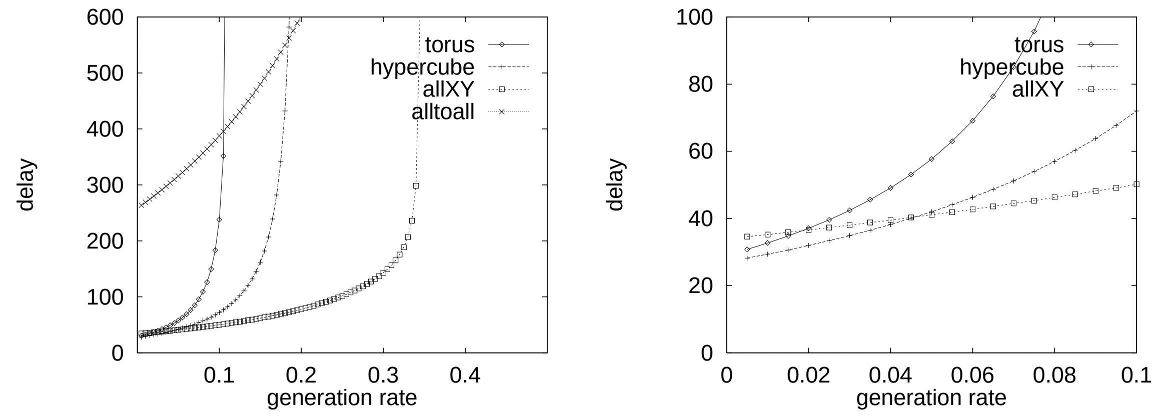


Figure 4.19: Packet delay as a function of packet generation rate ($\gamma = 1.0, N = 16$)

Figure 4.20 shows the impact of packet routing time on the average packet delay. The results are based upon a 16×16 torus network and a packet generation rate of 0.005. The packet routing speed has an impact on the delay for all topologies. For very small packet routing time ($\gamma = 0.25$), the torus topology has the smallest delay. When the packet routing time increases, the delay in torus increases drastically, while the delays in the all–to–all and allXY topologies increase slightly. In the all–to–all and allXY topologies a packet travels through fewer number of routers than it does in the torus topology. Hence the contention at routers does not affect the delay in the all–to–all and allXY topologies as much as it does in the torus and hypercube topologies. This study also implies that to achieve good packet delay for logical torus topology, fast routers are crucial, while a fast router is not as important in the all–to–all and allXY topologies.

Figure 4.21 shows the impact of network size on the packet delay for the topologies. The results are based upon a packet routing time of 1 time slot and a packet generation rate of 0.01. This figure shows the manner in which the delay time grows with regard to the network size. As discussed in section 2, ignoring network contention for a physical $N \times N$ torus, the all–to–all topology results in a packet delay of $O(\gamma + N^3)$, the allXY topology

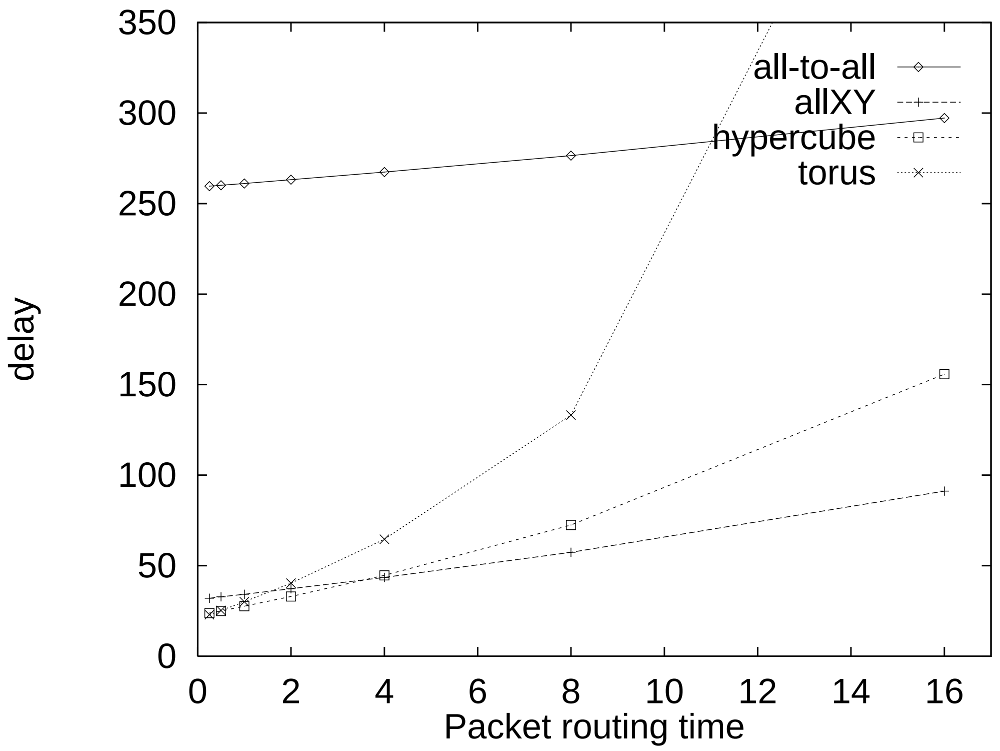
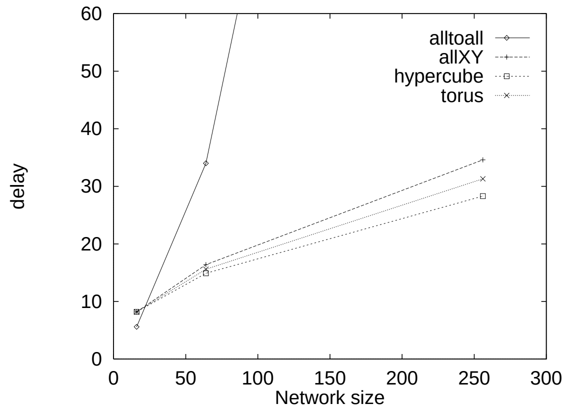


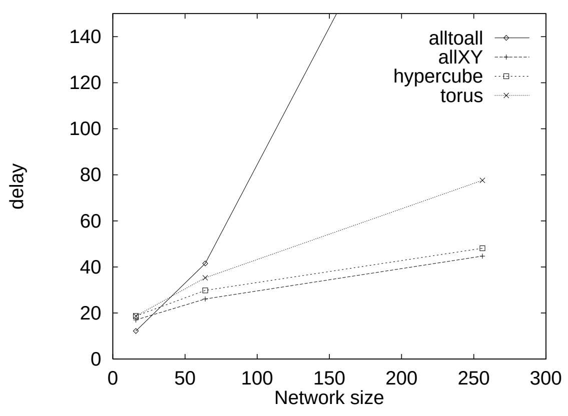
Figure 4.20: Impact of packet routing time on packet delay ($\lambda = 0.005, N = 16$)

has a delay of $O(\gamma + N^2)$, hypercube has a delay of $O(\gamma lg(N) + Nlg(N))$, and torus has a delay of $O(\gamma N)$. Thus, the all–to–all topology has very large delay when the network size is large. The delay differences among the other three topologies are relatively small for reasonably large sized networks. When the packet routing time is small ($\gamma = 1.0$), the hypercube topology scales slightly better than the torus and the allXY topologies as shown in Figure 4.21 (a). When γ is large ($\gamma = 4.0$), the hypercube topology and the allXY topology are better than the other two topologies as shown in Figure 4.21 (b).

From the above discussions, three parameters, N, γ and λ affect the average packet delay for all the logical topologies. Next, the regions in the (N, γ, λ) parameter space, where a logical topology has the lowest packet delay are identified. Figure 4.22 shows the best topologies in the parameter space (N, γ) with fixed λ. Comparing Figure 4.13, where the network contention is ignored, with Figure 4.22, where the contention is taken into consideration, it can be seen that the logical topologies with less connectivity suffer more from network contention. As can be seen from Figure 4.22 (a), with small packet generation rate, all four logical topologies occupy part of the (N, γ) parameter space, which indicates that under certain conditions, each of the four topologies out–performs the other three topologies. While in the case of large packet generation rate as shown in Figure 4.22 (b), the logical torus topology is pushed out of the best topology picture.

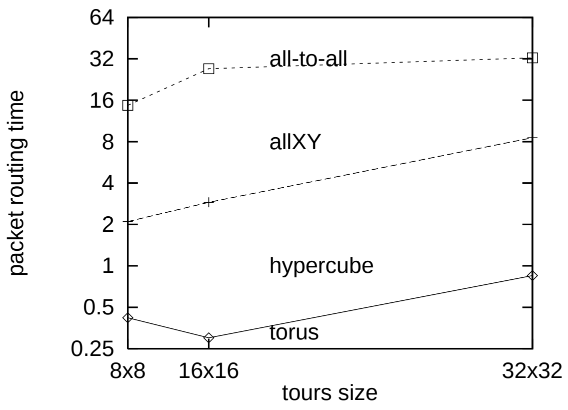


(a) $\gamma = 1.0$

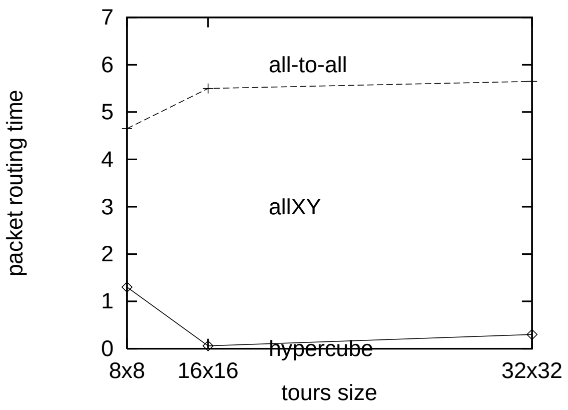


(b) $\gamma = 4.0$

Figure 4.21: impact of network size on the delay ($\lambda = 0.01$)



(a) $\lambda = 0.01$



(b) $\lambda = 0.06$

Figure 4.22: Best logical topology for a given packet generation rate

Figure 4.23 shows the best logical topologies on the (γ, λ) parameter space. Here, the underlying network is a 16×16 torus. Networks of different size exhibit similar characteristics. The majority of the (γ, λ) parameter space is occupied by the logical hypercube and allXY topologies. The logical torus topology is good only when the λ is small and γ is small. The logical all–to–all topology out–performs other topologies only when the network is almost saturated, that is, large λ or large γ. This indicates that in general, the logical hypercube and allXY topologies are better topologies than the logical torus and all–to–all topologies in terms of packet delay.

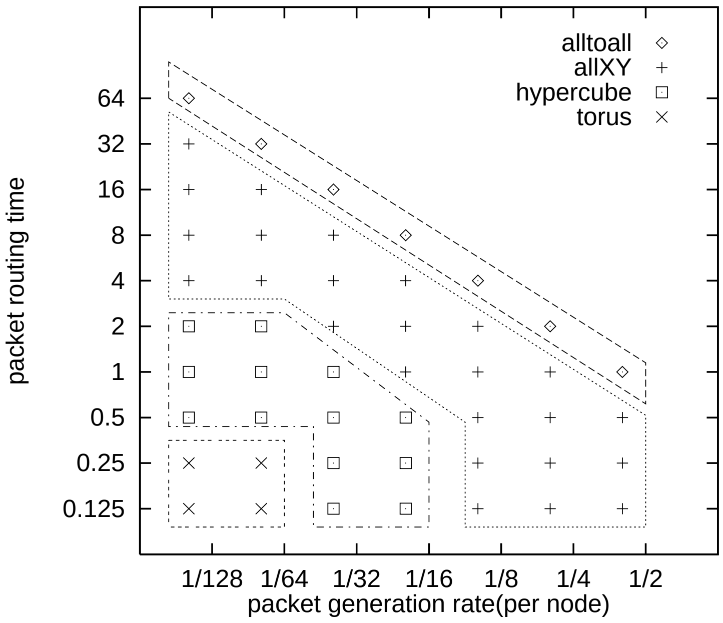


Figure 4.23: Best logical topology for a 16×16 torus

Figure 4.24 compares the performance of the logical hypercube and allXY topologies. Given a fixed γ, there is a packet generation rate, λ, above which the allXY topology out–performs the logical hypercube topology. When γ increases, the line in the figure moves down. In other words, the hypercube topology is more sensitive to the packet routing time γ.

4.5 Multi–hop communication vs single–hop communication

Previous sections considered the logical topologies that can be used to route packets and perform *multi–hop* communications. As discussed in Chapter 3, another way to perform dynamic communication on multiplexed optical networks is to use a path reservation algorithm which reserves an optical path from the source to the destination and then perform *single–hop* communications. The performance of these two communication

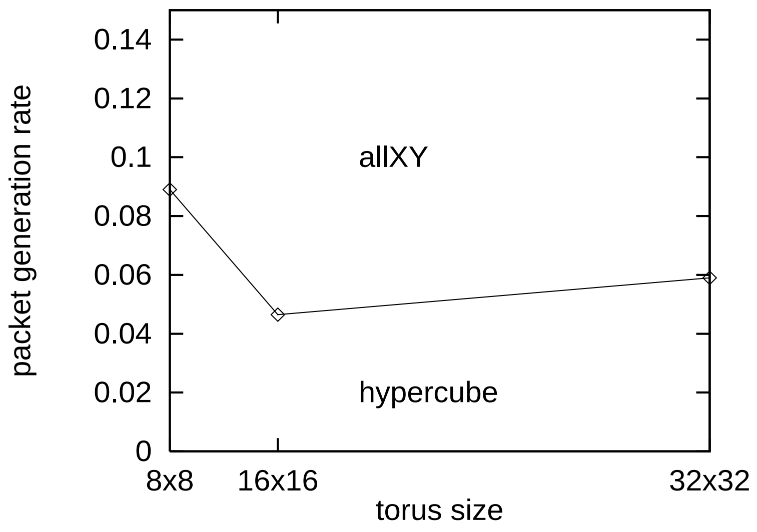


Figure 4.24: Best logical topology for a given packet routing time ($\gamma = 1.0$)

schemes on a physical 16×16 torus is compared in this section. The logical allXY topology is used as the logical topology for multi–hop communication since it offers large maximum throughput and reasonably small average package delay for this size of networks. To obtain a fair comparison, the following assumptions are made:

- Both networks have the same multiplexing degree. For a 16×16 torus, this means that both networks have a multiplexing degree of 32, which is required for the logical allXY topology.

- The data packet processing time in the multi–hop communication is equal to the control packet processing time in the path reservation algorithm, since electronic processing is involved in both cases. *Packet processing time*, γ, is used to represent both the data packet processing time in the multi–hop communication and the control packet processing time in the single–hop communication.

- Control packet propagation time between two neighboring nodes is equal to data packet propagation time between the source and the destination, which is equal to 1 time slot.

- It is assumed that a data *message* contains s packets. Accordingly, the *average message delay*, which is defined as the difference between the time the message is generated and the time when the whole message is received, is measured instead of the *average packet delay.* The notation λ_{msg} in this section represents the message generation rate per node per time slot. Since messages can be of different sizes, the *network load* is defined to be

$$network\ load = s \times \lambda_{msg} \times number\ of\ nodes,$$

which is equal to the total number of packets injected into the network. For the same reason, the throughput is measured in terms of packets delivered per time slot.

The analytical model for the multi–hop communication cannot model the communication performance when packets in a message are sent to the same destination since destinations of packets are no longer distributed uniformly among all nodes. In some sense the number of packets in a message reflects the locality of the communication traffic. All results in this section are obtained through simulations.

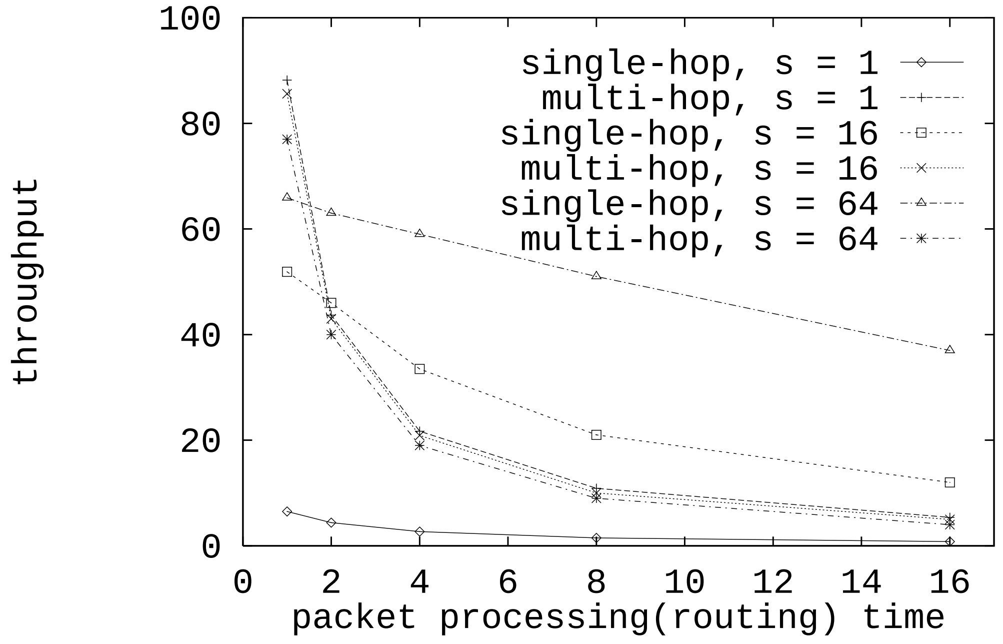
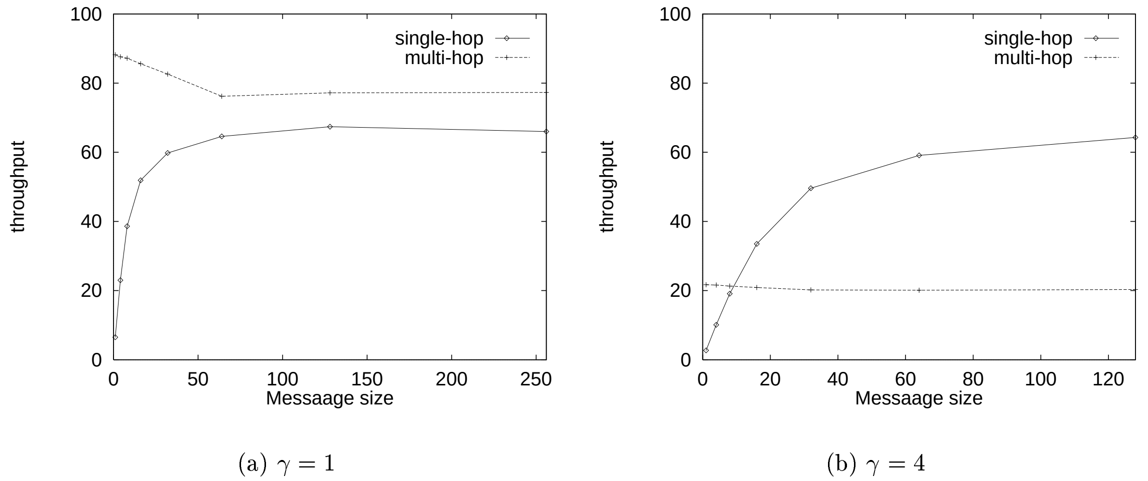


Figure 4.25: Maximum throughput

Figure 4.25 shows the maximum throughput of the two schemes with different message sizes, s, and packet processing times, γ. The packet processing time affects both the single–hop communication and the multi–hop communication, while the message size affects only single–hop communication (larger message size leads to higher maximum throughput). When the packet processing speed is fast, e.g. $\gamma = 1$, such that the path bandwidth is the bottleneck in the communication, the multi–hop communication offers larger maximum throughput than the single–hop communication. The reason is that multi–hop communication utilizes the links in the network more efficiently when the network is saturated and does not incur additional control overhead. However, the multi–hop communication is more sensitive to the packet processing time and the maximum throughput of the multi–

hop communication decreases drastically when the packet processing time increases. In the single–hop communication, the packet processing is only involved in the control network, thus, preserving the large bandwidth in the data network when the packet processing time is large. This effect manifests itself when the message size is reasonably large and the extra control overhead is amortized over the length of a message. Thus, the single–hop communication offers larger maximum throughput when the packet processing time is large and the message size is sufficiently large. Figures 4.26 (a) and 4.26 (b) show the maximum throughput with different message sizes for packet processing times of 1 and 4 respectively. As can be seen from the figures, when the packet processing time is small ($\gamma = 1$), the multi–hop communication offers larger maximum throughput for all message sizes. When the packet processing time is large ($\gamma = 4$), the single–hop communication has a larger maximum throughput when the message size is sufficiently large.



(a) $\gamma = 1$

(b) $\gamma = 4$

Figure 4.26: Maximum throughput for different message sizes

When the network is under light load, it is more meaningful to compare the message delay. Figure 4.27 shows the impact of the network load and the message size on the average message delay. In this figure, $\gamma = 1$. When the message size is small ($size = 4$), the multi–hop communication has smaller message delay. When the message size is large ($size = 64$), the single–hop communication offers smaller message delay. In both cases, the large network load amplifies the difference between single–hop and multi–hop communications. For messages of medium size ($size = 16$), the multi–hop communication has smaller delay when the network load is below a certain point. In general, small messages favor the multi–hop communication while large messages favor the single–hop communication.

The packet processing time affects the average message delay for both the single–hop communication and the multi–hop communication. In the single–hop communication,

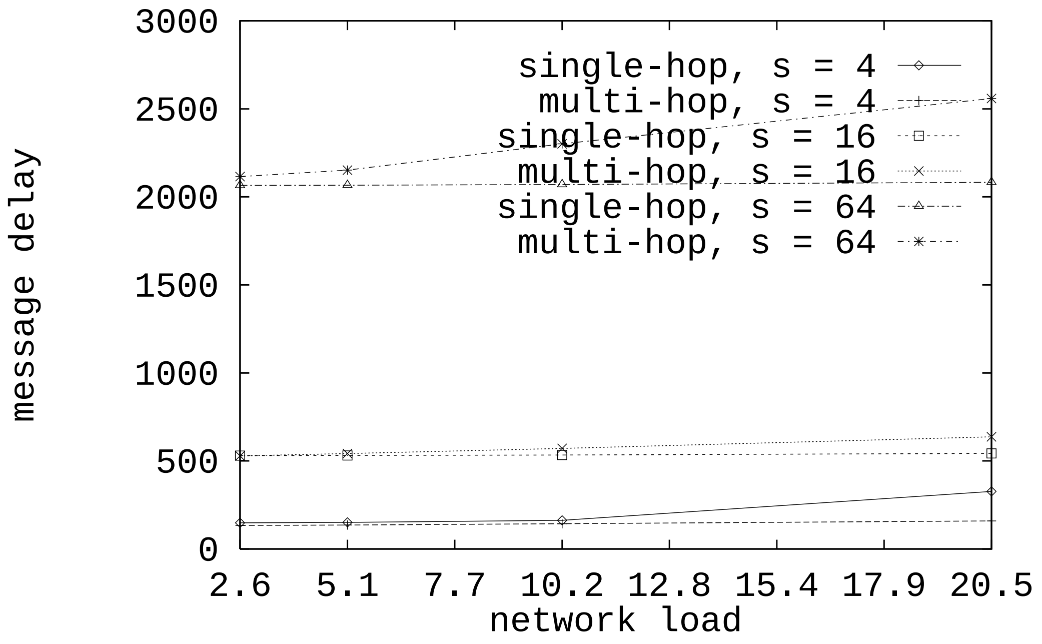


Figure 4.27: Impact of message size on the average message delay ($\gamma = 1$)

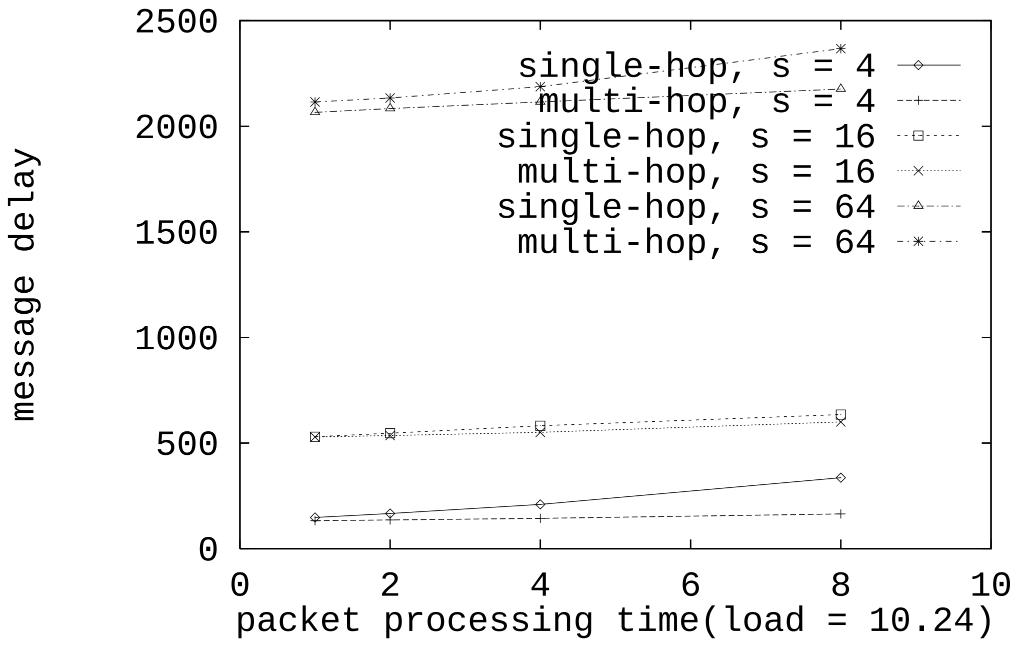


Figure 4.28: Impact of packet processing time on the average message delay ($\gamma = 1$)

the packet processing time affects the path reservation time only. Thus, given a fixed packet processing time, the extra control overhead is almost the same for all message sizes. In the multi–hop communication, the extra overhead applies to each packet in a message, and thus the larger the message size, the larger the overhead. Figure 4.28 shows the impact of the packet routing time on the average packet delay. In this figure, the same network load of 10.24 is considered for different message sizes (e.g. a generation rate of 0.01 for messages of size 4, $0.01 \times 4 \times 256 = 10.24$) with different message sizes. As can be seen from the figure, when the message size is small, the single–hop communication incurs larger message delay while for large message sizes, the multi–hop communication incurs larger message delay. The large packet processing time amplifies these effects.

4.6 Chapter summary

This chapter considered the logical topologies for routing message on top of torus topologies. Schemes for realizing the logical torus, hypercube, allXY (where all–to–all connections along each dimension are established) and all–to–all topologies on top of physical torus networks were discussed. Optimal schemes for realizing hypercube on top of physical arrays and rings were designed. Schemes that use at most 2 more channels than the optimal for realizing hypercube on top of meshes and tori were presented.

An analytical model for the maximum throughput and the packet latency for multi–hop networks was developed and verified through simulations. This analytical model was used to study the performance of the logical topologies and to identify the cases where each logical topology out–performs the other topologies. In general, the performance of the logical topologies with less connectivity, such as the torus and hypercube topologies, are more sensitive to the network load and the router speed while the logical topologies with more connectivity, such as the all–to–all and allXY topologies, are more sensitive to network size. Logical topologies with dense connectivity achieve higher maximum throughput than the topologies with less connectivity. In addition, they also scale better with regard to the network size. In terms of the maximum throughput, the topologies can be ordered as follows:

$$\textit{all–to–all} > allXY > \textit{hypercube} > \textit{torus}.$$

In term of the average packet delay, the logical torus topology achieves best results only when the router is fast and the network is under light load, while the logical all–to–all topology is best only when the router is slow and the network is almost saturated. In all other cases, logical hypercube and allXY topologies out–perform logical torus and all–to–all topologies. Comparing the logical allXY to the logical hypercube, the allXY topology is better when the network is under high load. These results hold for all network sizes.

This chapter further compared multi–hop communication with single–hop communication and identified the advantages and the limitations of each communication scheme. The study in this chapter used randomly generated communication traffic. Performance evaluation of these two schemes using communication patterns from real application programs, which confirms the results in this chapter, will be presented in Chapter 6. Multi–hop communication is more efficient than single–hop communication in terms of maximum throughput when the packet processing speed is not a bottleneck in the system and when the message size is small. When packet processing speed is slow, the single–hop communication has higher maximum throughput when the message size is sufficiently large. In terms of the average message delay when the network is under light load, large messages favor single–hop communication, while small messages favor multi–hop communication. The large packet processing time amplifies these effects. Table 4.3 and Table 4.4 summarize these conclusions.

Table 4.3: Maximum throughput on a 16×16 torus

	Small message size(4)	Large message size(64)
Small packet processing time	Multi–hop	Multi–hop
Large packet processing time	Multi–hop	Single–hop

Table 4.4: Average message delay on a 16×16 torus

Network load	Packet processing time	Message size		
		Small(4)	Medium(16)	Large(64)
Small	Small	Multi–hop	Multi–hop	Single–hop
	Large	Multi–hop	Single–hop	Single–hop
Large	Small	Multi–hop	Single–hop	Single–hop
	Large	Multi–hop	Single–hop	Single–hop

Both communication schemes suffer from the bottleneck of electronic processing, which occurs in the path reservation in single–hop communication and in the packet routing at intermediate nodes in multi–hop communication. Using the compiled communication technique discussed in the next chapter, this bottleneck can be removed.

Chapter 5
Compiled communication

In compiled communication, the compiler analyzes a program to determine its communication requirement. The compiler can then use the knowledge of the underlying architecture, together with the knowledge of the communication requirement, to manage network resources statically. As a result, runtime communication overheads, such as the path reservation overhead and the buffer allocation overhead, can be reduced or eliminated, and the communication performance can be improved. Due to the limited resources, the underlying network cannot support arbitrary communication patterns. Thus, compiled communication requires the compiler to analyze a program and partition the program into phases such that each phase has a fixed, pre-determined communication pattern that the underlying network can support. The compiler inserts code to reconfigure the network at phase boundaries, uses the knowledge of the communication requirement within each phase to manage network resources directly, and optimizes the communication performance.

A number of compiler issues must be addressed in order to apply the compiled communication technique to optical TDM networks. Specifically, given a multiplexing degree, the compiler must partition a program into phases such that each phase contains connections that can be realized by the underlying network with the given multiplexing degree. To obtain good performance, each phase must contain as much communication locality as possible so that less reconfiguration overhead will be incurred at runtime. A compiler, called the E-SUIF (extended SUIF) compiler, is implemented to support compiled communication. The structure of the compiler is shown in Figure 5.1. There are four major components in the system. The first component is the *communication analyzer* that analyzes a program and obtains its communication requirement on virtual processor grids. The second component is the *virtual to physical processor mapping* subsystem that computes the communication requirement of a program on physical processors. The third component is the *communication phase analysis* subsystem that partitions the program into phases such that each phase contains communications that the underlying network can

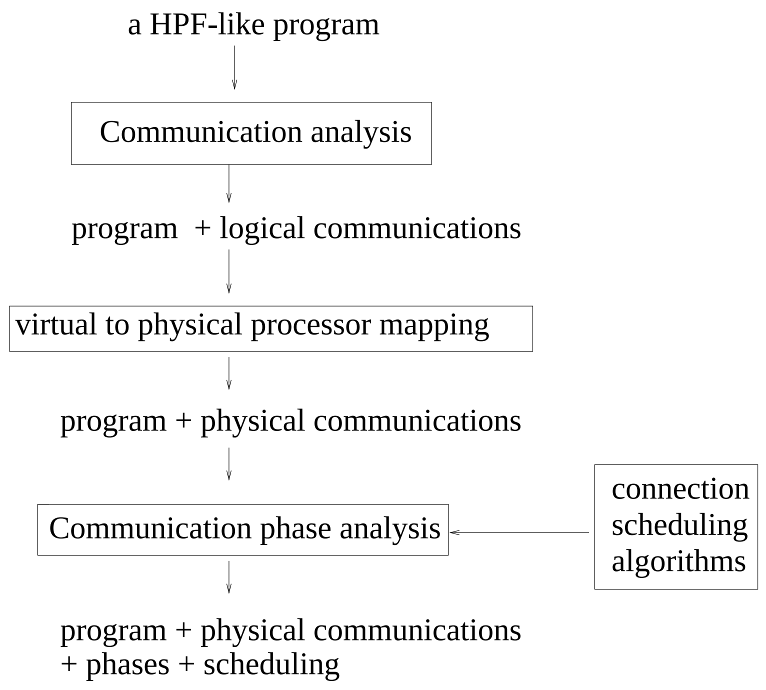

a HPF-like program
Communication analysis
program + logical communications
virtual to physical processor mapping
program + physical communications
Communication phase analysis
connection scheduling algorithms
program + physical communications + phases + scheduling

Figure 5.1: The major components in the E–SUIF compiler

support. The communication phase analysis utilizes a fourth component of the system, the *connection scheduling algorithms*, to realize a given communication pattern with a minimal number of channels.

Next, the programming model of the compiler will be discussed, followed by the four components needed to support compiled communication.

5.1 Programming model

The E–SUIF compiler considers structured HPF–like programs that contain conditionals and nested loops, but no arbitrary goto statements. The programmer explicitly specifies the data alignments and distributions. For simplicity, this chapter assumes that all arrays are aligned to a single virtual processor grid template, and the data distribution is specified through the distribution of the template. However, the implementation of the communication analyzer handles multiple virtual processor grids. Arrays are aligned to the virtual processor grid by simple affine functions. The alignments allowed are scaling, axis alignment and offset alignment. The mapping from a point $\vec{d}$ in data space to the corresponding point $\vec{e}$ on the virtual processor grid is specified by an alignment matrix M and an alignment offset vector $\vec{v}$. $\vec{e} = M\vec{d} + \vec{v}$. The alignment matrix M specifies the scaling and the axis alignment, thus it is a permutation of a diagonal matrix. The distribution of the virtual processor grid can be cyclic, block or block–cyclic. Assuming that there are p

processors in a dimension, and the block size of that dimension is b, the virtual processor e is in physical processor $e \ mod \ (p * b)/b$. For cyclic distribution, $b = 1$. For block distribution, $b = n/p$, where n is the size of the virtual processes along the dimension.

The communication analyzer performs communication optimizations on each subroutine. A subroutine is represented by an *interval flow graph* $G = (N, E)$, with nodes N and edges E. The communication optimizations are based upon a variant of Tarjan's intervals [75]. The optimizations require that there are no *critical edges* which are edges that connect a node with multiple outgoing edges to a node with multiple incoming edges. The critical edges can be eliminated by edge splitting transformation[32]. Figure 5.2 shows an example code and its corresponding interval flow graph.

```
      ALIGN (i, j) with VPROCS(i, j) :: x, y, z
      ALIGN (i, j) with VPROCS(2*j, i+1) :: w
(s1)  do i = 1, 100
(s2)     do j = 1, 100
(s3)        x(i,j)=...
(s4)     enddo
(s5)  enddo
(s6)  do i = 1, 100
(s7)     do j = 1, 100
(s8)        y(i,j)=w(i,j)
(s9)     enddo
(s10) enddo
(s11) do i = 1, 100
(s12)    do j = 1, 100
(s13)       z(i, j) = x(i+1, j)* w(i, ,j)
(s14)       z(i, j) = z(i, j)* y(i+1, ,j)
(s15)    end do
(s16)    w(i+1, 100) = ...
(s17) end do
```

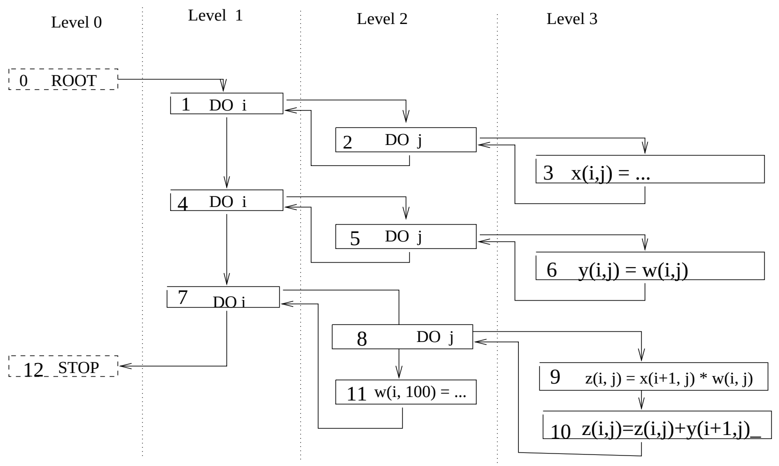


Figure 5.2: An example program and its interval flow graph

5.2 The communication analyzer

The communication analyzer analyzes the communication requirement on virtual processor grids and performs a number of common communication optimizations. This section presents the data flow descriptor used in the analyzer to describe communication, the general data flow algorithms to propagate the data flow descriptor, and the communication optimizations performed by the analyzer.

5.2.1 Section communication descriptor (SCD)

In order for the compiler to analyze the communication requirement of a program, data structures must be designed for the compiler to represent the communications in the program. The data structures must both be powerful enough to represent the communication requirement and simple enough to be manipulated easily.

The descriptor

The communication analyzer represents communication using *Section Communication Descriptor* (SCD). A $SCD = < A, D, CM, Q >$ consists of three components. The first component is the array region that is involved in the communication. This includes the array name A and the array region descriptor D. The second component is the communication mapping descriptor CM, which describes the source–destination relationship of the communication. The third component is a qualifier descriptor Q, which specifies the time when the communication is performed.

The *bounded regular section descriptor* (BRSD)[12] is used as the region descriptor. The region D is a vector of subscript values. Each element in the vector is either (1) an expression of the form $\alpha * i + \beta$, where α and β are invariants and i is a loop index variable, or (2) a triple $l : u : s$, where l, u and s are invariants. The triple, $l : u : s$, defines a set of values, $\{l, l+s, l+2s, ..., u\}$, as used in the array statement in HPF.

The source–destination mapping CM is denoted as $< src, dst, qual >$. The source, src, is a vector whose elements are of the form $\alpha * i + \beta$, where α and β are invariants and i is a loop index variable. The destination, dst, is a vector whose elements are of the form $\gamma * j + \delta$, where γ and δ are invariants and j is a loop index variable. The *mapping qualifier* list, $qual$, is a list of range descriptors. Each range descriptor is of the form $i = l : u : s$, where l, u and s are invariants and i is a loop index variable. The notation $qual = NULL$ and $qual = -$ denote that no mapping qualifier is needed. The mapping qualifier specifies the range of a variable in dst that does not occur in src to express the broadcast effect.

The qualifier Q is a range descriptor of the form $i = l : u : s$, where i is the loop index variable of the loop that directly encloses the SCD. This qualifier is used to indicate the iterations of the loop in which the SCD should be performed. If the SCD is to be performed in every iteration in the loop, $Q = NULL$ or $Q = -$. Q will be referred to as the *communication qualifier*. Notice that the qualifiers in most SCDs are NULL.

Operations on SCD

Operations, such as intersection, difference and union, on SCD descriptors are defined next. Since in many cases, operations do not have sufficient information to yield exact results, *subset* and *superset* versions of these operations are implemented. The analyzer uses a proper version to obtain conservative approximations. These operations are extensions of the operations on BRSD.

Subset Mapping testing. Testing whether a mapping is a subset of another mapping is one of the most commonly used operations in the analyzer. Testing that a mapping relation CM_1 ($=< s_1, d_1, q_1 >$) is a subset of another mapping relation CM_2 ($=< s_2, d_2, q_2 >$) is done by checking for a solution of equations $s_1 = s_2$ and $d_1 = d_2$, where variables in CM_1 are treated as constants and variables in CM_2 as variables, and subrange testing $q_1 \subseteq q_2$. Note that since the elements in s_1 and s_2 are of the form $\alpha * i + \beta$, the equations can generally be solved efficiently. Two mappings, CM_1 and CM_2 are *related* if $CM_1 \subseteq CM_2$ or $CM_2 \subseteq CM_1$. Otherwise, they are unrelated.

Subset SCD testing. Let $S_1 =< A_1, D_1, CM_1, Q_1 >$, $S_2 =< A_2, D_2, CM_2, Q_2 >$, $SCD_1 \subseteq SCD_2 \Longleftrightarrow A_1 = A_2 \wedge D_1 \subseteq D_2 \wedge CM_1 \subseteq CM_2 \wedge Q_1 \subseteq Q_2$.

Intersection Operation. The intersection of two SCDs represents the elements constituting the common part of their array sections that have the same mapping relation. The following algorithm describes the subset version of the intersection operation. Note that the operation requires the qualifier Q_1 to be equal to Q_2 to obtain a non empty result. ϕ denotes an empty set. This approximation will not hurt the performance significantly since most SCDs have $Q = -$.

$$
\begin{aligned}
&< A_1, D_1, CM_1, Q_1 > \cap < A_2, D_2, CM_2, Q_2 > \\
&= \phi, \text{ if } A_1 \neq A_2 \text{ or } CM_1 \text{ and } CM_2 \text{ are unrelated or } Q_1 \neq Q_2 \\
&= < A_1, D_1 \cap D_2, CM_1, Q_1 >, \text{ if } A_1 = A_2 \text{ and } CM_1 \subseteq CM_2 \text{ and } Q_1 = Q_2 \\
&= < A_1, D_1 \cap D_2, CM_2, Q_1 >, \text{ if } A_1 = A_2 \text{ and } CM_1 \supseteq CM_2 \text{ and } Q_1 = Q_2
\end{aligned}
$$

Difference Operation. The difference operation causes a part of the array region associated with the first operand to be invalidated at all the processors where it was available. In the analysis, the difference operation is only used to subtract elements killed (by a statement, or by a region), which means that the SCD to be subtracted always has $CM = \top$ and $Q = -$.

$< A_1, D_1, CM_1, Q_1 > - < A_2, D_2, \top, - >$
$= < A_1, D_1, CM_1, Q_1 >$, if $A_1 \neq A_2$
$= < A_1, D_1 - D_2, CM_1, Q_1 >$, if $A_1 = A_2$.

Union operation. The union of two SCDs represents the elements that can be in either part of their array section. This operation is given by:

$< A_1, D_1, CM_1, Q_1 > \cup < A_2, D_2, CM_2, Q_2 >$
$= < A_1, D_1 \cup D2, CM_1, Q_1 >$, if $A_1 = A_2$ and $CM_1 = CM_2$ and $Q_1 = Q_2$
$=$ list($< A_1, D_1, CM_1, Q_1 >$, $< A_2, D_2, CM_2, Q_2 >$), otherwise.

5.2.2 A demand driven array data flow analysis framework

Many communication optimization opportunities can be uncovered by propagating SCDs globally. For example, if a SCD can be propagated from a loop body to the loop header without being killed in the process of propagation, the communication represented by the SCD can be hoisted out of the loop body, that is, the communication can be vectorized. Another example is the redundant communication elimination. While propagating SCD_1, if SCD_2 is encountered such that SCD_2 is a subset of the SCD_1, then the communication represented by SCD_2 can be subsumed by the communication represented by SCD_1 and can be eliminated. Propagating SCDs backward can find the earliest point to place the communication, while propagating SCDs forward can find the latest point where the effect of the communication is destroyed. Both these two propagations are useful in communication optimizations. Since forward and backward propagation are quite similar, only backward propagation will be presented next.

Generic demand driven algorithms are developed to propagate SCDs through interval flow graph. The analysis technique is the reverse of the interval-analysis [30]. Specially, by reversing the information flow associated with program points, a system of request propagation rules is designed. SCDs are propagated until they cannot be propagated any further, that is, all the elements in the SCDs are killed. However, in practice, the compiler may choose to terminate the propagation prematurely to save analysis time while there are still elements in SCDs. In this case, since the analysis starts from the points that contribute to the optimizations, the points that are textually close to the starting points, where most of the optimization opportunities are likely to be present, are considered. This gives the demand driven algorithm the ability to trade precision for time. In the propagation, at a given time, only a single interval is under consideration. Hence, the propagations are logically done in an acyclic flow graph. During the propagation, a SCD may expand when it is propagated out of a loop. When a set of elements of SCD is killed inside a loop, the set is propagated into the loop to determine the exact point where the elements are killed. There are two types of propagations, *upward* propagation, in which SCDs may need to be expanded, and *downward* propagation, in which SCDs may need to be shrunk.

The format of a data flow *propagation request* is $< S, n, [UP|DOWN], level, cnum >$, where S is a SCD, n is a node in the flow graph, constants UP and $DOWN$ indicate whether the request is upward propagation or downward propagation, $level$ indicates at which level is the request and the value $cnum$ indicates which child node of n has triggered the request. A special value -1 for $cnum$ is used as the indication of the beginning of downward propagation. The propagation request triggers some local actions and causes the propagation of a SCD from the node n. The propagation of SCDs follows the following rules. It is assumed that node n has k children.

Propagation rules

RULE 1: upward propagation: regular node. The request on a regular node takes an action based on SCD set S and the local information. It also propagates the information upward. The request stops when S become empty. The rule is shown in the following pseudo code. In the code, functions *action* and *local* are depended on the type of optimization to be performed. The *pred* function finds all the nodes that are predecessors in the interval flow graph and the set $kill_n$ includes all the elements defined in node n. Note that $kill_n$ can be represented as an SCD.

```
request($< S_1, n, UP, level, 1 >$) $\wedge$ ... $\wedge$ request($< S_k, n, UP, level, k >$) :
  S = $S_1 \cap ... \cap S_k$
  action(S, local(n))
  if $(S - kill_n \neq \phi)$ then
    for all $m \in pred(n)$
      Let $n$ be $m$'s $j$th child
      request($< S - kill_n, m, UP, level, j >$)
```

A response to requests in a node n occurs only when all its successors have been processed. This guarantees that in an acyclic flow graph each node will only be processed once. The side effect is that the propagation will not pass beyond a branch point. A more aggressive scheme can propagate a request through a node without checking whether all its successors are processed. In that scheme, however, a nodes may need to be processed multiple times to obtain the final solution.

RULE 2: upward propagation: same level loop header node. The loop is contained in the current level. The request needs to obtain the summary information, K_n, for the interval, perform the action based on S and the summary information, propagate the information past the loop and trigger a downward propagation to propagate the information into the loop nest. Here, the summary function K_n, summarizes all the elements defined in the interval. It can be calculated either before hand or in a demand driven manner. The method to calculate the summary in a demand driven manner will be described later. Note that a loop header can only have one successor besides the entry edge into the loop body. The *cnum* of the downward request is set to -1 to indicate that it is the start of the downward propagation.

```
request($< S, n, UP, level, 1 >$):
  action(S, $K_n$)
  if ($S - K_n \neq \phi$) then
    for all $m \in pred(n)$
      Let $n$ be $m$'s $j$th child
      request($< S - K_n, m, UP, level, j >$)
  if ($S \cap K_n \neq \phi$) then
    request($< S \cap K_n, n, DOWN, level, -1 >$)
```

RULE 3: upward propagation: lower level loop header node. The relative level between the propagation request and the node can be determined by comparing the level in the request and the level of the node. Once a request reaches the loop header. The request will need to be expanded to be propagated in the upper level. At the same time, this request triggers a downward propagation for the set of elements that are killed in the loop. Assume that the loop index variable is i with bounds *low* and *high*.

```
request($< S, n, UP, level, 1 >$):
  calculate the summary of loop $n$
  outside = $expand(S, i, low : high) - \cup_{def} expand(def, i, low : high)$
  inside = $expand(S, i, low : high) \cap \cup_{def} expand(def, i, low : high)$
  if (outside $\neq \phi$) then
   for all $m \in pred(n)$
     Let $n$ be $m$'s $j$th child
     request($< outside, m, UP, level-1, j >$)
 if (inside $\neq \phi$) then
   request($< inside, n, DOWN, level, -1 >$)
```

The variable *outside* contains the elements that can be propagated out of the loop, while the variable *inside* contains the elements that are killed within the loop. The expansion function has the same definition as in [30]. For a SCD descriptor S, expand(S, k, low:high) is a function which replaces all single data item references $\alpha * k + \beta$ used in any array section descriptor D in S by the triple $(\alpha * low + \beta : \alpha * high + \beta : \alpha)$. The set def includes all the definitions that are the source of a flow-dependence.

RULE 4: downward propagation: lower level loop header node. This is the initial downward propagation. The loops index variable, i, is treated as a constant in the downward propagation. Hence, SCDs that are propagated into the loop body must be changed to be the initial available set for iteration i, that is, subtract all the variables killed in the iteration i+1 to high and propagate the information from the tail node to the head node. This propagation prepares the downward propagation into the loop body by shrinking the SCD for each iteration.

```
query($< S, n, UP, level, cnum >$):
  if ($cnum = -1$) then
    calculate the summary of loop $n$;
    request($< S - \cup_{def} expand(def, k, i+1 : high), l, DOWN, level-1, 1 >$);
  else
    STOP /* interval processed */
```

RULE 5: downward propagation: regular node. For regular node, the downward propagation is similar to the upward propagation.

```
request($< S_1, n, DOWN, level, 1 >$) $\wedge$ ... $\wedge$ request($< S_k, n, DOWN, level, k >$) :
  S = $S_1 \cap ... \cap S_k$
  action(S, local(n))
  if ($S - kill_n \neq \phi$) then
    for all $m \in pred(n)$
    Let $n$ be $m$'s $j$th child
    request($< S - kill_n, m, DOWN, level, j >$)
```

RULE 6: downward propagation: same level loop header node. When downward propagation reaches a loop header (not the loop header whose body is being processing), it must generate further downward propagation request to go deeper into the body.

```
request($< S, n, DOWN, level, 1 >$):
  action(S, summary(n));
  if ($S - K_n \neq \phi$) then
    for all $m \in pred(n)$
      Let $n$ be $m$'s $j$th child
      request($< S - K_n, m, DOWN, level, j >$);
  if ($S \cap K_n \neq \phi$) then
    request($< S \cap K_n, n, DOWN, level, -1 >$);
```

Summary calculation

During the request propagation, the summary information of an interval is needed when a loop header is encountered. An algorithm is described to obtain the summary information in a demand driven manner. The calculation of kill set of the interval is used as an example. Let $kill(i)$ be the variables killed in node i, K_{in} and K_{out} be the variables killed before and after the node respectively. Figure. 5.3 depicts the demand driven algorithm. The algorithm propagates the data flow information from the tail node to the header node in the interval using the following data flow equation:

$$K_{out}(n) = \cup_{s \in succ(n)} K_{in}(s)$$
$$K_{in}(n) = kill(n) \cup K_{out}(n)$$

When an inner loop header is encountered, a recursive call is issued to get the summary information for the inner interval. Once a loop header is reached, the kill set needs to be expanded to be used by the outer loop.

```
(1)     Summary_kill(n)
(2)       $K_{out}(tail) = \phi$
(3)       for all $m \in T(n)$ and level(m) = level(n)-1 in backward order
(4)         if m is a loop header then
(5)           $K_{out}(m) = \cup_{s \in succ(m)} K_{in}(s)$
(6)           $K_{in}(m)$ = summary_kill(m) $\cup K_{out}(m)$
(7)         else
(8)           $K_{out}(m) = \cup_{s \in succ(m)} K_{in}(s)$
(9)           $K_{in}(m) = kill(m) \cup K_{out}(m)$
(10)      return (expand($K_{in}(header)$, i, low:high))
```

Figure 5.3: Demand driven summary calculation

5.2.3 The analyzer

The analyzer performs message vectorization, redundant communication elimination and communication scheduling using algorithms based upon the demand driven algorithms described in the previous section. The analyzer performs the following steps:

1. *Initial SCD calculation.* Here the analyzer calculates the communication requirement for each statement that contains remote memory references. Communications required by each statement are called *initial SCDs* for the statement and are placed preceding the statement.

2. *Message vectorization and available communication summary calculation.* The analyzer propagates initial SCDs to the outermost loops in which they can be placed. In addition to message vectorization optimization, this step also calculates the summary of communications that are available after each loop. This information is used in the next step for redundant communication elimination.

3. *Redundant communication elimination.* The analyzer performs redundant communication elimination using a demand driven version of availability communication analysis [30], which computes communications that are available before each statement. A communication in a statement is redundant if it can be subsumed by available communications at the statement. The analyzer also eliminates partially redundant communications.

4. *Message scheduling.* The analyzer schedules messages within each interval by placing messages with the same communication patterns together and combining the messages to reduce the number of messages.

Initial SCD Calculation

The *owner computes* rule is assumed which requires each remote item referenced on the right handside of an assignment statement to be sent to the processor that owns the left handside variable. Initial SCDs for each statement represent this data movement. Since the ownership of array elements determines communication patterns, the ownership of array elements will be described before the initial SCD calculation step is presented.

Ownership.

All arrays are aligned to a single virtual processor grid by affine functions. The alignments allowed are scaling, axis alignment and offset alignment. The mapping from a point $\vec{d}$ in data space to a corresponding point $\vec{e}$ on the virtual processor grid (the owner of $\vec{d}$) can be specified by an alignment matrix M and an alignment offset vector $\vec{v}$ such that $\vec{e} = M\vec{d} + \vec{v}$. Using the alignment matrix and the offset vector, the owner of a data element can be determined. Consider the array w in the example program in Figure 5.2, the alignment matrix and the offset vector are given below.

$$M_w = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix}, \ \vec{v}_w = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Initial SCD Calculation.

Using the ownership information, the initial SCDs are calculated as follows. Let us consider each component in an initial $SCD =< A, D, CM, Q >$. A is the array to be communicated. The region D contains a single index given by the array subscript expression. The qualifier $Q =-$ since initial communications must be performed in every iteration. Let $CM =< src, dst, qual >$. Since initially communication does not perform broadcast, $qual =-$. Hence, the calculation of src and dst, which will be discussed in the following text, is the only non-trivial computation in the calculation of initial SCDs.

Let $\vec{i}$ be the vector of loop induction variables. When subscript expressions are affine functions, an array reference can be expressed as $A(G\vec{i} + \vec{g})$, where A is the array name, G is a matrix and $\vec{g}$ is a vector. G is called the *data access matrix* and $\vec{g}$ the *access offset vector.* The data access matrix, G, and the access offset vector, $\vec{g}$, describe a mapping from a point in the iteration space to a point in the data space. Let G_l, $\vec{g}_l$, M_l, $\vec{v}_l$ be the data access matrix, the access offset vector, the alignment matrix and the alignment vector for the *lhs* array reference, and G_r, $\vec{g}_r$, M_r, $\vec{v}_r$ be the corresponding quantities for the *rhs* array reference. The source processor src and destination processor dst are given by:

$$src = M_r(G_r\vec{i} + \vec{g}_r) + \vec{v}_r, \qquad dst = M_l(G_l\vec{i} + \vec{g}_l) + \vec{v}_l$$

Consider the communication of $w(i,j)$ in statement $s13$ in Figure 5.2. The analyzer can obtain from the program the data access matrices, access offset vectors, alignment matrices and alignment vectors and from them the SCD for the communication given below. As an indication of the complexity of a SCD, the structure for this communication required 524 bytes to store.

$$M_z = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ \vec{v}_z = \begin{pmatrix} 0 \\ 0 \end{pmatrix},\ M_w = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix},\ \vec{v}_w = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$G_l = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ \vec{g}_l = \begin{pmatrix} 0 \\ 0 \end{pmatrix},\ G_r = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ \vec{g}_r = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$< A = w, D = (i,j), CM =< (2*j, i+1), (i,j), ->, Q =->$$

Message Vectorization and Available Communication Summary

In this phase, the analyzer computes *backward exposed* communications, which are SCDs that can be hoisted out of a loop, and *forward exposed* communications, which are SCDs that are available after the loop. Backward exposed communications represent actual communications vectorized from inside the loop. When a SCD is vectorized, the initial SCD at the assignment statement are replaced by SCDs for backward exposed communications at loop headers. Forward exposed communications represent the communications that are performed inside a loop and are still alive after the loop. Hence they can be used to subsume communications appearing after the loop. By using data dependence information, backward and forward exposed communications are calculated by propagating SCDs from inner loop bodies to loop headers using a simplified version of the rules discussed in previous section.

Algorithms for the forward and backward exposed communication calculation are described in Figure 5.4 (a) and (b). Since only UP propagation is needed, $Request(S,n)$ is used to denote placing a propagation of S after node n. In the algorithms, S is a SCD occurring inside the interval whose header is node n and whose induction variable is i with lower bound 1 and upper bound h, $anti\_def$ is the set of definitions in the interval that have anti–dependence relation with the original array reference that causes the communication S, $flow\_def$ is the set of definitions in the interval that have flow–dependence relation with the original array reference that causes the communication S. For a SCD, S, $expand(S,i,1:h)$ first determines which portion of the $S =< A, D, CM, Q >$ to be expanded. If D is to be expanded, that is, i occurs in D, the function will replace all single data item references $\alpha * i + \beta$ used in D by the triple $\alpha + \beta : \alpha * h + \beta : \alpha$. If D cannot be expanded, that is, after expansion D is not in the allowed form, then the communications will stay inside the loop. If $CM =< src, dst, qual >$ is to be expanded, that is, i occurs in dst but not in src and D, the function will add $i = 1 : h : 1$ into the mapping qualifier list $qual$.

The algorithms determine the part of communications $Outside$, that can be hoisted out of a loop, and $Inside$, that cannot be hoisted out of the loop. In forward exposed communication calculation, the analyzer makes $Outside$ as the forward exposed communication and ignores the $Inside$ part. In backward exposed communication calculation, the analyzer makes $Outside$ as backward exposed communication. In addition, the analyzer must also change the original SCD according to contents of $Inside$. In the case when the SCD can be fully vectorized, the SCD in the original statement is removed. In the case when the SCD cannot be fully vectorized, part of the communication represented by $Outside$ is hoisted out of the loop, while other part represented by $Inside$ stays at the original statement. Thus, the SCD in the original statement must be modified by a communication qualifier to indicate that the SCD only remains in iterations that generate communications in $Inside$.

```
request($S, n$) :
  $Outside = expand(S, i, 1 : h) -$
    $\cup_{anti\_def} expand(anti\_def, i, 1 : h)$
  if ($Outside \neq \phi$) then
    record $Outside$ as
      forward exposed in node n
    Let m be the header of the
      interval including node $n$
    $request(Outside, m)$;
```

(a) Forward exposed communication

```
request($S, n$) :
  $Outside = expand(S, i, 1 : h) -$
    $\cup_{flow\_def} expand(flow\_def, i, 1 : h)$
  $Inside = expand(S, i, 1 : h) \cap$
    $\cup_{flow\_def} expand(flow\_def, i, 1 : h)$
  if ($Outside \neq \phi$) then
    convert $Inside$ in terms of $S$
      with qualifier, denoted as $D$
    if (conversion not successful) then
      stop /* fail */
    else
      change the S into D
      record $Outside$ as backward
        exposed comm. at node $n$.
      Let m be the header of the
        interval including node $n$
      $request(Outside, m)$;
```

(b) Backward exposed communication

Figure 5.4: Algorithms for the forward and backward exposed communication

Consider communications in the loop in Figure 5.5. Assume that arrays a, b and d are identically aligned to the virtual processor grid, initial SCDs, $C1$ and $C2$, are shown in Figure 5.5. $C3$, $C4$ and $C5$ are the communications after the backward exposed communication calculation. Calculating the backward exposed communication for $C1$ results in communication $C3$ in the loop header and the removal of the communication $C1$ from its original statement. Calculating the backward exposed communication for $C2$ puts $C4$ in the loop header and changes $C2$ into $C5$. Note that, there is a flow–dependence relation from b(i) to b(i-1). In calculating the backward exposed communication for SCD $C2$,

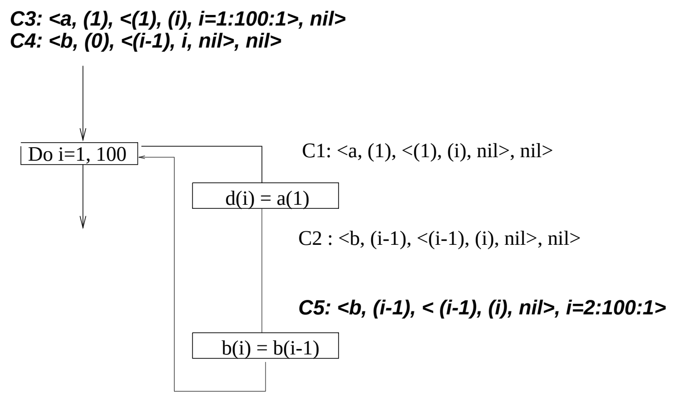

C3: <a, (1), <(1), (i), i=1:100:1>, nil>
C4: <b, (0), <(i-1), i, nil>, nil>
Do i=1, 100
C1: <a, (1), <(1), (i), nil>, nil>
d(i) = a(1)
C2 : <b, (i-1), <(i-1), (i), nil>, nil>
C5: <b, (i-1), < (i-1), (i), nil>, i=2:100:1>
b(i) = b(i-1)

Figure 5.5: Calculating backward exposed communications

$Inside =< b, (1:99:1), < (i-1, 1), (i, 1), -->, -->$. Converting $Inside$ back in terms of $C2$ results in $C5$.

Redundant Communication Elimination

This phase calculates available communications before each statement, and eliminates a communication at the statement if the communication is available. This optimization is done by propagating SCDs forward until all elements are killed. During the propagation, if another SCD that can be subsumed is encountered, that SCD is redundant and can be eliminated.

```
request($S_1, n, UP$) $\wedge$ ...
$\wedge$request($S_k, n, UP$) :
  S = $S_1 \cap ... \cap S_k$
  if (SCDs in n is a subset of S) then
    remove the SCDs
  if ($S - kill_n \neq \phi$) then
    for all $m \in succ(n)$
      request($S - kill_n, m, UP$)
```

(a) Actions on nodes within an interval

```
request($S, n, UP$):
  calculate the summary of loop $n$
  Inside= $expand(S, i, 1 : i-1) \cap$
    $(\cup_{def} expand(def, i, 1 : i-1))$
  if (inside $\neq \phi$) then
    Let $l$ be the first node.
    request($Inside, l, DOWN$)
```

(b) Actions on a loop header

Figure 5.6: Actions in forward propagation

Using the interval analysis technique [30], two passes are needed to obtain the data flow solutions in an interval. Initially, UP propagations are performed. Once the UP propagations reach interval headers, summaries of the SCDs are calculated and DOWN propagations of the summaries are triggered. Note that since the data flow effect of propagating SCDs between intervals is captured in the message vectorization phase of the analyzer, both the UP and DOWN propagations are performed within an interval in this phase.

Assuming that node n has k predecessors. When propagating SCDs within an interval in forward propagation, actions in a node will be triggered only when all its predecessors place requests. The nodes calculate the SCD available by performing intersection on all SCDs that reach it, check whether communications within the node can be subsumed, and propagate the live communications forward. Figure 5.6 (a) describes actions on the nodes inside the interval in an UP forward propagation. When the UP propagation reaches an interval boundary, the summary information is calculated by obtaining all the elements that are available in iteration i, and a DOWN propagation is triggered. Note that in forward propagation, communications can be safely assumed to be performed in every iteration ($Q =-$), since the effect of the communication must guarantee that the valid values are at the proper processors for the computation. Figure 5.6 (b) shows actions at interval boundaries. The propagation of a DOWN request is similar to that of an UP request except that a DOWN propagation stops at interval boundaries.

Global Message Scheduling

After the redundant communication elimination phase, the analyzer further reduces the number of messages using a global message scheduling algorithm proposed by Chakrabarti et al. in [14]. The idea of this optimization is to combine messages that are of the same communication pattern into a single message to reduce the number of messages in a program. In order to perform message scheduling, the analyzer first determines the earliest and latest points for each communication. Placing the communication in any point between the earliest and the latest points that dominates the latest point always yields correct programs. Thus, the analyzer can schedule the placement of messages such that messages of same communication patterns are placed together and are combined to reduce the number of messages.

The latest point for a communication is the place of the SCD after redundant communication elimination. Note that after message vectorization, SCDs are placed in the outermost loops that can perform the communications. The earliest point for a SCD can be found by propagating the SCD backward. As in [14], it is assumed that communication for a SCD is performed at a single point. Hence, the backward propagation will stop after an assignment statement, a loop header or a branch statement where part of the SCD is killed. Since the propagation of SCDs stops at a loop header node, only the UP propagation is needed. Once the earliest and latest points for each communication are known, the greedy heuristic in [14] is used to perform the communication scheduling.

5.2.4 Evaluation of the analyzer

The analyzer is implemented as part of the E–SUIF compiler which is developed to support compiled communication on optical TDM networks. The E–SUIF compiler is based on the Stanford SUIF compiler [73]. The generation of a program used for evaluations is carried out in the following steps. First, a sequential program is compiled using SUIF frontend, *scc*, to generate the SUIF intermediate representation. Next, the SUIF transformer, *porky*, is used to perform a number of scalar optimizations including copy propagation, dead code elimination and induction variable elimination. The communication preprocessing phase is used to annotate global arrays with data alignment information. The analyzer is then invoked to analyze and optimize communications in the program. After communication optimizations, the backend of the compiler inserts a library call into the SUIF intermediate representation for each SCD remaining in the program. Finally, the *s2c* tool is used to convert the SUIF intermediate representation into C program, which is the one that is executed for evaluation.

To evaluate performance of the analyzer, a communication emulation system is developed. The system takes SCDs as input, emulates the communications described by the SCDs and collects statistics about the required communications, such as the total number of elements communicated and the total number of messages communicated. The emulation system provides an interface to C program in the form of a library call whose arguments include all information in a SCD. The compiler backend in E–SUIF automatically generates the library call for each SCD remaining in the program. In this way, the communication performance of a program can be evaluated in the emulation system by running programs generated by the E–SUIF compiler.

Six programs, L18, ARTDIF, TOMCATV, SWIM, MGRID and ERHS are used in the experiment. Programs ARTDIF, TOMCATV, SWIM, MGRID and ERHS are from the SPEC95 benchmark suite. The descriptions of the programs are as follows.

1. L18 is the explicit hydrodynamics kernel in livermore loops (loop 18).

2. ARTDIF is a kernel routine obtained from HYDRO2D program, which is an astrophysical program for the computation of galactical jets using hydrodynamical Navier Stokes equations.

3. TOMCATV does the mesh generation with Thompson's solver.

4. SWIM is the SHALLOW weather prediction program.

5. MGRID is the simple multigrid solver for computing a three dimensional potential field.

6. ERHS is part of the APPLU program, which is the solver for five coupled parabolic/elliptic partial differential equations.

Table 5.1 shows the analysis cost of the analyzer. The analyzer, which implements all the optimization algorithms on all SCDs in the programs, was run on a SPARC 5 machine with 32MB memory. Row 2 and Row 3 shows the program sizes. Row 4 shows the cumulative memory requirement, which is the sum of number of SCDs passing through each node. This number is approximately equal to the memory requirement of traditional data flow analysis. The value in parenthesis is the maximum number of cumulative SCDs in a node, which is the extra memory needed by the analyzer. In the analyzer, the size of a SCD ranges from 0.6 to about 3 kbytes. The results show that traditional analysis method will require large amount of memory when a program is large, while the analyzer uses little extra memory. Row 5 gives the raw analysis times and row 6 shows the rate at which the analyzer operates in units of *lines/sec*. On an average, the analyzer compiles 172 lines per second for the six programs. Row 9 shows the total time, which includes analysis time and the time to load and store the SUIF structure, for reference. In most cases, the analysis time is only a fraction of the load and store time.

Program	L18	ARTDIF	TOMCATV	SWIM	MGRID	ERHS
size(lines)	83	101	190	429	486	1104
# of initial SCDs	35	12	108	76	125	403
accu. memory req.	348(1)	175(1)	5078(3)	767(1)	1166(1)	6029(5)
analysis time(sec)	0.62	0.32	3.47	1.87	1.92	20.92
lines / sec	133	316	54	229	253	52
total time(sec)	2.00	1.75	6.95	6.65	12.52	35.42

Table 5.1: Analysis time

Table 5.2 and Table 5.3 show the effectiveness of the optimizations in the analyzer. Table 5.2 shows the reduction of the total number of elements to be communicated and Table 5.3 shows the reduction of the total number of messages. Both cyclic and block distributions on 16 PE systems are considered. This experiment is conducted using the test input provided by the SPEC95 benchmark for programs TOMCATV, SWIM, MGRID ERHS. The outermost iteration number in MGRID is reduced to 1 (from 40). Problem sizes of 6×100 for L18 and 402×160 for ARTDIF are used. The number of elements and number of messages communicated after all optimizations is compared to those after message vectorization optimization. Table 5.2 shows that for cyclic distribution, an average reduction of 31.5% of the total communication elements is achieved. The block distribution greatly reduces the number of elements to be communicated and affects the optimization performance of the analyzer. For block distribution, the average reduction is 23.1%. Table 5.3 shows that the analyzer reduces the total number of messages by 36.7% for cyclic distribution and by 35.1% for block distribution. These results indicate that global communication optimization opportunities are quite common and the analyzer developed is effective in finding these opportunities.

Dist.	Opt.	L18 $\times 10^4$	ARTDIF $\times 10^5$	TOMCATV $\times 10^8$	SWIM $\times 10^7$	MGRID $\times 10^7$	ERHS $\times 10^6$
cyclic	Vector.	1.38	7.01	1.38	6.38	5.69	3.62
	Final	0.96	5.73	0.34	4.58	5.69	2.29
		69.6%	81.7%	24.6%	71.8%	100%	63.3%
		$\times 10^3$	$\times 10^4$	$\times 10^6$	$\times 10^6$	$\times 10^6$	$\times 10^6$
block	Vector.	3.26	7.17	5.74	3.38	8.49	3.11
	Final	2.57	6.97	5.12	1.08	8.49	1.65
		78.8%	97.2%	89.1%	32.0%	100%	53.1%

Table 5.2: Total number of elements to be communicated

Dist.	Opt.	L18	ARTDIF	TOMCATV	SWIM	MGRID	ERHS
cyclic	Vector.	368	400	68555	3892	17662	1.14×10^6
	Final	96	336	41075	1807	17662	0.72×10^6
		26.1%	84.0%	59.9%	46.4%	100%	63.1%
block	Vector.	330	185	16750	3894	14650	9.20×10^5
	Final	90	161	10915	2209	14650	4.89×10^5
		27.3%	87%	65.2%	56.7%	100%	53.2%

Table 5.3: Total number of messages

5.3 Virtual to physical processor mapping

In order to support compiled communication, communication patterns on physical processors must be computed. This section assumes that the physical processor grid has the same number of dimensions as the logical processor grid. Notice that this is not a restriction because a dimension in the physical processor grid can always be collapsed by assigning a single processor to that dimension. This section presents algorithms to compute communications on physical processors from SCDs. The computation may not always be precise due to symbolic constants in the SCD that are unknown at compile time. The algorithms employ multi–level approximation schemes to obtain best information.

Given a $SCD =< A, D, CM =< src, dst, qual >, Q >$, let us first consider the case where A is an one-dimensional array and the virtual processor grid is also one-dimensional. Let $src = \alpha * i + \beta$ and $dst = \gamma * i + \delta$, $\alpha \neq 0$, $\gamma \neq 0$, and $qual = NULL$. $qual \neq NULL$ will be considered later when multi-dimensional arrays and multi-dimensional virtual processor grids are discussed. Let the alignment matrix and the offset vector be M_A and v_A, that is, element A[n] is owned by virtual processor $M_A * n + v_A$. Let us assume that the number of physical processors is p and the block size of the distribution of virtual processor grid is b. For an element A[n], the physical source processor of the communication can be computed as follows.

$$(M_A * n + v_A) \ mod \ (p * b)/b$$

The virtual destination processor can be computed by first solving the equation

$$(M_A * n + v_A) = \alpha * i + \beta \text{ to obtain } i = (M_A * n + v_A - \beta)/\alpha$$

and then replacing the value of i in dst to obtain the virtual destination processor $\gamma * (M_A * n + v_A - \beta)/\alpha + \delta$. Thus, the physical destination processor is given by

$$(\gamma * (M_A * n + v_A - \beta)/\alpha + \delta) \ mod \ (p * b)/b.$$

The physical communication pattern for the SCD can be obtained by considering all elements in D. However, there are situations that the exact region D cannot be determined at compile time. It is desirable to have a good approximation scheme that computes the communication patterns when D cannot be determined at compile time.

Before the approximation scheme is presented, let us first examine the relation between communications on physical processors and that on virtual processors. Let us use notation $src \rightarrow dst$ to represent a communication from src to dst. Given a data region $D = l : u : s$, the communications on virtual processors can be derived as follows. By mapping D to the virtual processor grid, the source processors of the communications can be obtained. Since the mapping from data space to the virtual processor grid is linear, the set of source processors can be represented as a triple $vs_l : vs_u : vs_s$, that is, the source processors on the virtual processor grid are vs_l, $vs_l + vs_s$, $vs_l + 2 * vs_s$, ..., vs_u. Due to the way in which $CM.src = \alpha * i + \beta$ is computed, equations $vs_l + i * vs_s = CM.src$, $i = 0, 1, 2, ...$, always have integer solutions. Since $CM.dst$ is of the form $\gamma * i + \delta$, where γ and δ are constants, the destination processors on the virtual processor grid can also be represented as a triple $vd_l : vd_u : vd_s$, where $vd_l = \gamma * ((vs_l - \beta)/\alpha) + \delta$, $vd_u = \gamma * ((vs_u - \beta)/\alpha) + \delta$ and $vd_s = \gamma * vs_s/\alpha$. Notice that because of the way in which CM is computed, all the division operations in the formula result in integers. Thus, communications on the virtual processor grid can be represented as $vs_l \rightarrow vd_l : vs_u \rightarrow vd_u : vs_s \rightarrow vd_s$, meaning the set

$$\{vs_l \rightarrow vd_l, vs_l + vs_s \rightarrow vd_l + vd_s, ..., vs_u \rightarrow vd_u\}.$$

Communications on physical processors are obtained by mapping virtual processors onto physical processors. Given a block–cyclic distribution with block size b and processor number p, a sequence of processors on the virtual processor grid $l, l + s, l + 2 * s, ...$ will be

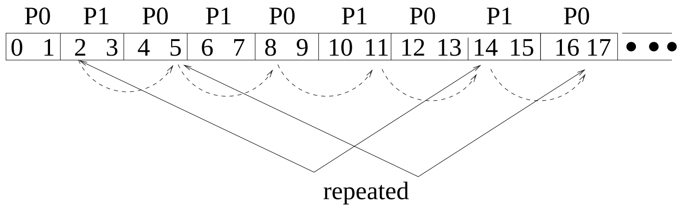


Figure 5.7: Virtual processor space

mapped to a sequence of physical processors repeatedly. For example, assuming that $p = 2$ and $b = 2$, the sequence of virtual processors $2, 2 + 3 = 5, 2 + 2 * 3 = 8, 2 + 3 * 3 = 11, ...$ will be mapped to physical processors $1, 0, 0, 1$ repeatedly as shown in Figure 5.7. As will be seen later, this characteristic can be utilized to develop an approximation algorithm for the cases when D is unknown at compile time. A point e in the virtual processor grid can be represented by two components (pp, o), where $pp = e \ mod \ (p * b)/b$ is the physical processor that contains e and $o = e \ mod \ b$ is the offset of e within the processor. Let (pp_k, e_k) correspond to $l + k * s$, $k = 0, 1, ...$ It can be easily shown that

$$pp_i = pp_j \wedge e_i = e_j \text{ implies } pp_{i+1} = pp_{j+1} \wedge e_{i+1} = e_{j+1}$$

Since in the (pp, o) space, there are p choices for pp and b choices for o, Thus, there exists a k, $k \leq p * b$, such that $pp_k = p_0$ and $e_k = e_0$, which determines a repetition point. In the previous example, consider the sequence

$$2 = (1, 0), 5 = (0, 1), 8 = (0, 0), 11 = (1, 1), 14 = (1, 0)....$$

Thus, the physical processors repeat the sequence $1, 0, 0, 1$.

Communications on physical processor contains two processors, the source processor and the destination processor. Thus, in order for the communications to repeat, both source and destination processors must repeat. Following the above discussion, the communication on the virtual processor grid, $src \rightarrow dst$, can be represented by four components (spp, so, dpp, do), where spp is the physical processor that contains src, so is the offset of src within the processor, dpp is the physical processor that contains dst, do is the offset of dst within the processor. Assuming that the source array and the destination array are mapped to the same virtual processor grid, there are p choices for spp and dpp, and b choices for so and do. Thus, there exists k, $k \leq p^2 b^2$, such that both source and destination processors, and thus the communication pattern, will repeat themselves. The following lemma summarizes these results. Using this lemma, communication patterns can be obtained by considering the elements in D until the repetition point or the end of D, whichever occurs first.

Lemma: Assume that the virtual processor grid is distributed over p processors with block size b. Let $SCD =< A, D = l : u : s, CM =< src, dst, qual >, Q >$, assuming u

```
Compute_1–dimensional_pattern($D$, $CM.src$, $CM.dst$)

  Let $D = l : u : s$, $CM.src = \alpha * i + \beta$, $CM.dst = \gamma * i + \delta$
  **if** ($l$ contains variables) **then**
    **return** all–to–all connections
  **end if**
  **if** ($\alpha$, $\beta$, $\gamma$ or $\delta$ are variables) **then**
    **return** all–to–all connections
  **end if**
  $pattern = \phi$
  **for each** element $i$ in $D$ **do**
    $pattern = pattern + communication\ of\ i$
    **if** (communication repeated) **then**
      **return** $pattern$
    **end if**
  **end for**
```

Figure 5.8: Algorithm for 1-dimensional arrays and 1-dimensioanl virtual processor grid

is infinite, there exist a value k, $k \leq p^2 b^2$, such that the communication for all $m \geq k$, $A[l + m * s]$ has the same source and destination as the communication for $A[l + (m - k) * s]$.
Proof: Follows from above discussions. □

The implication of the lemma is that the algorithm to determine the communication pattern for the SCD can stop when the repetition point occurs. In other words, when the upper bound of D is unknown, the communication pattern can be approximated by using the repetition point. Figure 5.8 shows the algorithm to compute the physical communication pattern for a 1–dimensional array and a 1–dimensional virtual processor grid. The algorithm first checks the SCD. Let $D = l : u : s$ and $CM =< \alpha + \beta * i, \gamma + \delta * i, - >$. If l contains variables or the mapping is not clean (α, β, γ or δ are symbolic constants), the communication is approximated with all–to–all connections. Note that by the semantics of array sections, when l is unknown, the compiler cannot determine the actual sequence of elements in an array section. When s contains variables, it will be approximated by 1, that is, D is approximated by a superset $l : u : 1$. When u contains variables, the physical communication is approximated by considering all elements until the repetition point. Note that when u contains a variable, the sequence in D is l, $l + s$, $l + 2 * s$, ... Although the upper bound of the sequence is unknown to the compiler, the repetition point can be used to approximate the communication pattern.

Now let us consider multi-dimensional arrays and multi–dimensional virtual processor grids. In an n–dimensional virtual processor grid, a processor is represented by a

n–dimensional coordinate $(p_1, p_2, ..., p_n)$. The algorithm to compute the communication pattern finds all pairs of source and destination processors that require communication. This is done by considering the dimensions in virtual processor grid one at a time. A set of $src = (sp_1, sp_2, ..., sp_n) \rightarrow dst = (dp_1, dp_2, ..., dp_n)$ pairs is used to represent the communications. A wild–card, $*$, is used to represent the dimension within a tuple that has not been considered. Initially the communication set contains a single element where all dimensions are wild–cards. When one dimension is considered, it generates a 1-dimensional communication pattern for a specific dimension in the source and the destination, denoted as *src_dim* and *dst_dim* respectively. This 1-dimensional pattern may degenerate to contain only source processors or destination processors. A cross product operation is defined to merge the 1-dimensional communication patterns into the n-dimensional communication. This operation is similar to the cross product of sets except that specific dimensions are involved in the operation. For the degenerate form of the 1-dimensional pattern, the operation only involves source processors or destination processors.

For example, consider the communication for

$SCD = < y, (1:4:1, 1:4:1), < src = (i,j), dst = (j,i), qual = NULL >, NULL >$.

Further assume that the virtual processor grid is distributed on 2 processors with block size of 2 in each dimension and array y is identically mapped to the virtual processor grid. Initially, the communication set contains a single element $(*,*) \rightarrow (*,*)$, indicating that all dimensions in the source and destination processor have not been considered. Considering the first dimension in the data space, which is identically mapped to the first dimension of the virtual grid. Hence, $src\_dim = 1$. From the mapping relation $CM.src$ and $CM.dst$, it is can found that dimension 2 in the destination processor correspond to dimension 1 in the source processor. Hence, $dst\_dim = 2$. Applying the algorithm for the 1–dimensional communication pattern obtains the communication to be $\{0 \rightarrow 0, 1 \rightarrow 1\}$ with $src\_dim = 1, dst\_dim = 2$. Taking the cross product of this pattern with the 2-dimensional communication set $\{(*,*) \rightarrow (*,*)\}$ yields $\{(0,*) \rightarrow (*,0), (1,*) \rightarrow (*,1)\}$. Considering the second dimension of the data space, the 1–dimensional communication set is $\{0 \rightarrow 0, 1 \rightarrow 1\}$ with $src\_dim = 2, dst\_dim = 1$. Taking the cross product of this pattern set to the 2–dimensional communication set gives $\{(0,0) \rightarrow (0,0), (0,1) \rightarrow (1,0), (1,0) \rightarrow (0,1), (1,1) \rightarrow (1,1)\}$, which is the physical communication for the SCD.

The above example does not take constant mappings and non–NULL qualifiers into consideration. The algorithm to compute communication patterns for multi-dimensional arrays that is shown in Figure 5.9 considers all these situations. The algorithm first checks whether the mapping relation can be processed. If one loop induction variable occurs in two or more dimensions in $CM.src$ or $CM.dst$, the algorithm cannot find the correlation between dimensions in source and destination processors, and the communication pattern for the SCD is approximated by all–to–all connections. If the SCD passes the mapping relation test, the algorithm determines for each dimension in the data space the corresponding dimension sd in the source processor grid. If it does not exist, the data dimension is not distributed and need not be considered. If there exists such a dimension, the algorithm then tries to find the corresponding dimension dd in the destination processor grid by checking whether there is a dimension dd such that $CM.dst[dd]$ contains the same looping index variable as the source dimension $CM.src[sd]$. If such dimension exists, the algorithm computes 1-dimensional communication pattern between dimension sd in the source processor and dimension dd in the destination processor, then cross–products the 1-dimensional communication pattern into the n-dimensional communication pattern. When dd does not exist, the algorithm determines a degenerate 1-dimensional pattern, where only source processors are considered, and cross-products the degenerate 1-dimensional pattern into the communication pattern. After all dimensions in the data space are considered, there may still exist dimensions in the source processor (in the virtual processor grid) that have not been considered. These dimensions should be constants and are specified by the alignment matrix and the alignment offset vector. The algorithm fills in the constants in the source processors. Dimensions in destination processor may not be fully considered, either. When $CM.qual \neq NULL$, the algorithm finds for each item in $CM.qual$ the corresponding dimension, computes all possible processors in that dimension and cross–products the list into the communication list. Finally, the algorithm fills in all constant dimensions in the destination.

An example in Figure 5.10 illustrates how communications on physical processors are derived. In the program, the virtual processor grid is 3-dimensional and the alignment array and the alignment offset vector for arrays x and y are as follows:

$$M_x = \begin{pmatrix} 0 & 2 \\ 1 & 0 \\ 0 & 0 \end{pmatrix}, \; \vec{v}_x = \begin{pmatrix} 0 \\ 2 \\ 1 \end{pmatrix} M_y = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \; \vec{v}_y = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}.$$

Let us assume that the virtual processor grid, $VPROCS$, is distributed as $p = (2, 2, 1)$, which means 2 processors in dimension 0, 2 processors in dimension 1 and 1 processor in dimension 2, and $b = (2, 2, 1)$, which means the block size 2 in dimension 0, 2 in dimension 1 and 1 in dimension 2. After communication analysis, the SCD to represent the

```
Compute communication pattern(SCD)
  Let $SCD =< A, D, CM, Q >$
  **if** (the format of $CM$ is not good) **then**
    **return** all-to-all connections
  **end if**
  $pattern = \{(*, *, ..., *)\}$
  **for each** dimension $i$ in the array **do**
    Let $sd$ be the corresponding dimension in source processor grids.
    Let $dd$ be the corresponding dimension in destination processor grids.
    1dpattern = compute_1-dimensional_pattern($D[i]$, $CM.src[sd]$, $CM.dst[dd]$)
    pattern = cross_product(pattern, 1dpattern)
  **end for**
  pattern = source_processor_constants(pattern)
  **for each** element $i$ in the mapping qualifier **do**
    Let $dd$ be the corresponding destination processor dimension.
    1dpattern = compute_1-dimensional_pattern($CM.qual[i]$, $-$, $CM.dst[dd]$)
    pattern = cross_product(pattern, 1dpattern)
  **end for**
  pattern = destination_processor_constants(pattern)
  **return** pattern
```

Figure 5.9: Algorithm for multi–dimensional array

```
ALIGN (i, j) with VPROCS(2*j, i+2, 1) :: x
ALIGN (i) with VPROCS(1, i+1, 2) :: y
DO i = 1, 5
DO j = 1, 5
    x(i, j) = y(i) + 1
END DO
END DO
```

Figure 5.10: An example

communication is as follows:

$SCD =< y, (1:5:1), < src = (1, i+1, 2), dst = (2*j, i+2, 1), qual = \{j = 1:5:1\} >, NULL >.$

The communication on physical processors is computed as follows. First consider the dimension 0 in the array y. From the alignment, the algorithm knows that dimension 1 in the virtual processor grid corresponds to this dimension in the data space. Checking dst in M, the algorithm can find that dimension 1 in destination corresponds to dimension 1 in source processors. Applying the 1-Dimensional mapping algorithm, an 1–dimensional communication pattern $\{0 \rightarrow 1, 1 \rightarrow 0\}$ with $src\_dim = 1$ and $dst\_dim = 1$ is obtained. Thus the communication list becomes $\{(*, 1, *) \rightarrow (*, 0, *), (*, 0, *) \rightarrow (*, 1, *)\}$ after taking the cross product with the 1–dimensional pattern. Next, the other dimensions in source processors, including dimension 0 that is always mapped to processor 0 and dimension 2 that is always mapped to processor 1 are considered. After filling in the physical processor in these dimensions in source processors, the communication pattern becomes $\{(0, 1, 1) \rightarrow (*, 0, *), (0, 0, 1) \rightarrow (*, 1, *)\}$. Considering the $qual$ in M, the dimension 0 of the destination processor can be either 0 or 1. Applying the cross product operation, the new communication list $\{(0, 1, 1) \rightarrow (0, 0, *),\ (0, 1, 1,) \rightarrow (1, 0, *),\ (0, 0, 1) \rightarrow (0, 1, *),\ (0, 0, 1) \rightarrow (1, 1, *)\}$ is obtained. Finally, the dimension 2 in the destination processor is always mapped to processor 0, Thus, the final mapping is $\{(0, 1, 1) \rightarrow (0, 0, 0), (0, 1, 1s) \rightarrow (1, 0, 0), (0, 0, 1) \rightarrow (0, 1, 0), (0, 0, 1) \rightarrow (1, 1, 0)\}$.

There are several levels of approximations in the algorithm. First, when the algorithm cannot correlate the source and destination processor dimensions from the mapping relation, the algorithm uses an approximation of all–to–all connections. If the mapping relation contains sufficient information to distinguish the relation of the source and destination processor dimension, computing the communication pattern for a multi-dimensional array reduces to computing 1-dimensional communication patterns, thus the approximations within each dimension are isolated to that dimension and will not affect the patterns in other dimensions. Using this multi-level approximation scheme, some information is obtained when the compiler does not have sufficient information for a communication.

5.4 Connection scheduling algorithms

Once the communication requirement on physical processors is obtained, the compiler uses off–line algorithms to perform connection scheduling and determines the communication phases in a program. This section presents the connection scheduling algorithms and their performance evaluation. These algorithms assume a torus topology.

For a given network, a set of connections that do not share any link is called a configuration. In an optical TDM network with path multiplexing, multiple configurations can be supported simultaneously. Specifically, for a network with multiplexing degree d, d configurations can be established concurrently. Thus, for a given communication pattern, realizing the communication pattern with a minimum multiplexing degree is equivalent to determining the minimum number of configurations that contain all the connections in the pattern. Next, some definitions will be presented to formally state the problem of connection scheduling. A connection from a source src to a destination dst is denoted as (src, dst).

A pair of connections (s_1, d_1) and (s_2, d_2) are said to **conflict**, if they cannot be simultaneously established because they use the same link.

A **configuration** is a set of connections $\{(s_1, d_1), (s_2, d_2), ..., (s_m, d_m)\}$ such that no connections in the set conflict.

Given a set of connections $Comm = \{(s_1, d_1), (s_2, d_2), ..., (s_m, d_m)\}$, the set $MC = \{C_1,$ C_2, ..., C_t $\}$ is a **minimal configuration set** for $Comm$ iff:

- each $C_i \in MC$ is a configuration and each connection $(s_i, d_i) \in R$ is contained in exactly one configuration in MC; and
- each pair of configurations $C_i, C_j \in MC$ contain connections $(s_i, d_i) \in C_i$ and $(s_j, d_j) \in C_j$ such that (s_i, d_i) conflicts with (s_j, d_j).

It has been shown that optimal message scheduling for arbitrary topologies is NP-complete [19]. Therefore these algorithms are heuristics that are demonstrated to provide good performance. Three connection scheduling heuristic algorithms that compute a minimal configuration set for a given connection set $Comm$ are described next.

5.4.1 Greedy algorithm

In the greedy algorithm, a configuration is created by repeatedly putting connections into the configuration until no additional connection can be established in that configuration. If additional connections remain, another configuration is created and this process is repeated till all connections have been processed. This algorithm is a modification of an algorithm proposed in [61]. The algorithm is shown in Figure 5.11. The time complexity of the algorithm is $O(|Comm| \times max_i(|C_i|) \times d)$, where $|Comm|$ is the number of the connections, $|C_i|$ is the number of connections in configuration C_i and d is the number of configurations generated.

For example consider the linearly connected nodes shown in Figure 5.12. The result for applying the greedy algorithm to schedule connections set $\{(0,\ 2),\ (1,\ 3),(3,\ 4),$

```
(1)   MC = $\phi$, k = 1
(2)   repeat
(3)     $C_k = \phi$
(4)     foreach $(s_i, d_i) \in Comm$
(5)       if $(s_i, d_i)$ does not conflict with any connection in $C_k$ then
(6)         $C_k = C_k \bigcup\ (s_i, d_i)$
(7)         Comm = Comm $-\ (s_i, d_i)$
(8)       end if
(9)     end for
(10)    MC = MC $\bigcup\ C_k$
(11)  until $Comm = \phi$
```

Figure 5.11: The greedy algorithm.

(2, 4)} is shown in Figure 5.12(a). In this case, (0, 2) will be in time slot 1, (1, 3) in time slot 2, (3, 4) in time slot 1 and (2, 4) in time slot 3. Therefore, multiplexing degree 3 is needed to establish the paths for the four connections. However, as shown in Figure 5.12 (b), the optimal scheduling for the four connections, which can be obtained by considering the connection in different order, is to schedule (0, 2) in slot 1, (1, 3) in slot 2, (3, 4) in slot 2 and (2, 4) in slot 1. The second assignment only use 2 time slots to establish all the connections.

Figure 5.12: Scheduling connections (0, 2), (1, 3),(3, 4), (2, 4)

5.4.2 Coloring algorithm

The greedy algorithm processes the connections in an arbitrary order. This subsection describes an algorithm that applies a heuristic to determine the order to process the connections. The heuristic assigns higher priorities to connections with fewer conflicts. By giving the connections with less conflicts higher priorities, each configuration is likely to accommodate more connections and thus the multiplexing degree needed for the patterns is likely to decrease.

The problem of computing the minimal configuration set is formalized as a graph coloring problem. A coloring of a graph is an assignment of a color to each node of the graph in such a manner that no two nodes connected by an edge have the same color. A conflict graph for a set of connections is built in the following manner, (1) each node in the graph corresponds to a connection and (2) an edge is introduced between two nodes if the connections represented by the two nodes are conflicted. As stated by the theorem given below, the number of colors used to color the graph is equal to the number of configurations needed to handle the connections.

Theorem: Let $Comm = \{(s_1, d_1), (s_2, d_2), ..., (s_m, d_m)\}$ be the set of connections and $G = (V, E)$ be the conflict graph for $Comm$. There exists a configuration set $M = \{C_1, C_2, ..., C_t\}$ for R if and only if G can be colored with t colors.

Proof: Since connections that correspond to the nodes with the same color do not conflict with each other, they can be placed in one configuration. $\square$

Thus, the coloring algorithm attempts to minimize the number of colors used in coloring the graph. Since the coloring problem is known to be NP-complete, a heuristic is used for graph coloring. The heuristic determines the order in which nodes are colored using the node priorities. The algorithm is summarized in Fig 5.13. It should be noted that after a node is colored, the algorithm updates the priorities of uncolored nodes. This is because in computing the degree of an uncolored node, only the edges that connect the node to other uncolored nodes are considered. The algorithm finds a solution in linear time (with respect to the size of the conflict graph). The time complexity of the algorithm is $O(|Comm|^2 \times max_i(|C_i|) \times d)$, where $|Comm|$ is the number of the connections, $|C_i|$ is the number of connections in configuration C_i and d is the total number of configurations generated.

For torus and mesh networks, a suitable choice for priority for a connection is the ratio of the number of links in the path from the source to the destination and the degree of the node corresponding to the connection in G. Applying the coloring algorithm to the example in Figure 5.12, in the first iteration, the connections are reordered as $\{(0, 2), (1, 3), (2, 4), (3, 4)\}$ and connections (0, 2), (2, 4) will be put in time slot 1. In the second iteration, connections (1, 3), (3, 4) are put in time slot 2. Hence, applying the coloring algorithm will use 2 time slots to accommodate the connections.

5.4.3 Ordered AAPC algorithm

The graph coloring algorithm has better performance than the greedy heuristic. However, for dense communication patterns the heuristics cannot guarantee that the multiplexing degree found would be bounded by the minimum multiplexing degree needed to realize the all-to-all pattern. The algorithm described in this section targets dense com-

```
(1)   Construct conflict graph G = (V, E)
(2)   Calculate the priority for each node
(3)   MC = $\phi$, k = 1
(4)   NCSET = V
(5)   **repeat**
(6)     Sort NCSET by priority
(7)     WORK = NCSET
(8)     $C_k = \phi$
(9)     **while** (WORK $\neq \phi$)
(10)      Let $n_f$ be the first element in WORK
(11)      $C_k = C_k \bigcup \{< s_f, d_f >\}$
(12)      NCSET = NCSET $-\{n_f\}$
(13)      **for each** $n_i \in NCSET$ and $(f, i) \in E$ **do**
(14)        update the priority of $n_i$
(15)        WORK = WORK - $\{n_i\}$
(16)      **end for**
(17)    **end while**
(18)    MC = MC + $\{C_k\}$
(19)  **until** NCSET = $\phi$
```

Figure 5.13: The graph coloring heuristic.

munication patterns. By grouping the connections in a more organized manner, better performance can be achieved for dense communication.

The worst case of arbitrary communication is the *all-to-all personalized communication* (AAPC) where each node sends a message to every other node in the system. Any communication pattern can be embedded in AAPC. Many algorithms [33, 38] have been designed to perform AAPC efficiently for different topologies. Among these algorithms, the ones that are of interests to us are the phased AAPC algorithms, in which the AAPC connections are partitioned into contention–free phases. A phase in this kind of AAPC corresponds to a configuration. Some phased AAPC algorithms are optimal in that every link is used in each phase and every connection follows the shortest path. Since all the connections in each AAPC phase are contention–free, they form a configuration that uses all the links in the system. Each phase in the phased AAPC communication forms an *AAPC configuration.* The set of *AAPC configurations* for AAPC communication pattern is called *AAPC configurations set.* The following theorem states the property of connection scheduling using AAPC phases.

Theorem: Let $Comm = \{(s_1, d_1), (s_2, d_2), ..., (s_m, d_m)\}$ be the set of connections, if $Comm$ can be partitioned into K phases $P_1 = \{(s_1, d_1), ..., (s_{i_1}, d_{i_1})\}$,
$P_2 = \{(s_{i_1+1}, d_{i_1+1}), ..., (s_{i_2}, d_{i_2})\}$, ... , $P_K = \{(s_{i_{K-1}+1}, d_{i_{K-1}+1}), ..., (s_{i_K}, d_{i_K})\}$, such that

P_i, $1 \leq i \leq K$, is a subset of an AAPC configuration. Using the greedy algorithm to schedule the connections $(s_1, d_1), (s_2, d_2), ..., (s_m, d_m)$ results in a multiplexing degree less than or equal to K.

Proof: The theorem is proven by contradiction that for any α, $1 \leq \alpha \leq m$, let $(s_\alpha, d_\alpha) \in P_\beta$, $1 \leq \beta \leq K$, connections (s_1, d_1), ..., (s_α, d_α) can be scheduled by the greedy algorithm using a multiplexing degree less than or equal to β.

Let $(s_\alpha, d_\alpha) \in P_\beta$ be the first connection that does not satisfy the above proposition. That is, (s_1, d_1), ..., $(s_{\alpha-1}, d_{\alpha-1})$ are scheduled using a multiplexing degree of β and (s_α, d_α) cannot be accommodated in configuration β. Since the connections in P_β do not conflict with each other, another connection that belongs to P_γ, $\gamma < \beta$ must be scheduled in configuration β. Hence, (s_α, d_α) is not the first connection that does not satisfy the proposition, which contradicts the assumption. □

The theorem states that if the connections are reordered by the AAPC phases, at most all AAPC phases are needed to realize arbitrary pattern using the greedy scheduling algorithm. For example, following the algorithms in [33], $N^3/8$ phases are needed for a $N \times N$ torus. Therefore, in a $N \times N$ torus, $N^3/8$ degree is enough to satisfy any communication pattern.

To obtain better performance on dense communication patterns, it is better to keep the connections in their AAPC format as much as possible. It is therefore better to schedule the phases with higher link utilization first. This heuristic is used in the ordered AAPC algorithm. In ordered AAPC algorithm, the rank of the AAPC phases is calculated so that the phase that has higher utilization has higher rank. The phases are then scheduled according to their ranks. The algorithm is depicted in Figure 5.14. The time complexity of this algorithm is $O(|Comm|(lg(|Comm|)+max_i(|C_i|)\times K))$, where $|Comm|$ is the number of the connections, $|C_i|$ is the number of connections in configuration C_i and K is the number of configurations needed. The advantage of this algorithm is that for this algorithm the multiplexing degree is bounded by $N^3/8$. Thus, in situations where the greedy or coloring heuristics fail to meet this bound, AAPC can be used.

5.4.4 Performance of the scheduling algorithms

In this section, the performance of the connection scheduling algorithms on 8×8 torus topology is studied. The performances of the algorithms are evaluated using randomly generated communication patterns, patterns encountered during data redistribution, and some frequently used communication patterns. The metric used to compare the algorithms is the multiplexing degree needed to establish the connections. It should be noted that a

```
(1)  PhaseRank[*] = 0
(2)  for$(s_i, d_i) \in Comm$ do
(3)      let $(s_i, d_i) \in A_k$
(4)      PhaseRank[k] = PhaseRank[k] + length($(s_i, d_i)$)
(5)  end for
(6)  sort phase according to PhaseRank
(7)  Reorder $Comm$ according the sorted phases.
(8)  call greedy algorithm
```

Figure 5.14: Ordered AAPC scheduling algorithm

dynamic scheduling algorithm will not perform better than the greedy algorithm since it must establish the connections by considering the connections in the order that they arrive.

A *random communication pattern* consists of a certain number of random connections. A random connection is obtained by randomly generating a source and a destination. Uniform probability distribution is used to generate the sources and destinations. The *data redistribution communication patterns* are obtained by considering the communication results from array redistribution. In this study, data redistributions of a 3D array are considered. The array has block–cyclic distribution in each dimension. The distribution of a dimension can be specified by the block size and the number of processors in the dimension. A distribution is denoted as *p:block(s)*, where p is the number of processors in the distribution and s is the block size. When the distribution of an array is changed (which may result from the changing of the value p or s), communication may be needed. Many programming languages for supercomputers, such as CRAFT FORTRAN, allow an array to be redistributed within a program.

Table 5.4 shows the multiplexing degree required to establish connections for random communication patterns using the algorithms presented. The results in each row are the averages obtained from scheduling 100 different randomly generated patterns with the specific number of connections. The results in the column labeled *combined algorithm* are obtained by using the minimum of the coloring algorithm and the AAPC algorithm results. Note that in compiled communication, more time can be spent to obtain better runtime network utilization. Hence, the combined algorithm can be used to obtain better result by the compiler. The percentage improvement shown in the sixth column is achieved by the combined algorithm over the dynamic scheduling. It is observed that the coloring algorithm is always better than the greedy algorithm and the AAPC algorithm is better than the other algorithms when the communication is dense. It can be seen that for sparse random pat-

number of connections.	greedy algorithm	coloring algorithm	AAPC algorithm	combined algorithm	improvement percentage
100	7.0	6.7	6.9	6.6	6.3%
400	16.5	16.1	16.5	15.9	3.8%
800	27.2	25.9	26.5	25.6	6.3%
1200	36.3	34.5	35.3	34.2	6.1%
1600	45.0	43.5	43.4	42.8	5.1%
2000	53.4	50.4	50.4	49.7	7.4%
2400	60.8	57.5	57.4	56.7	7.2%
2800	68.8	64.4	62.4	62.4	10.2%
3200	76.3	70.8	64	64	19.2%
3600	83.9	76.8	64	64	31.1%
4000	91.6	83	64	64	43.1%

Table 5.4: Performance for random patterns

terns (100 - 2400 connections), the improvement range varies from 3.8% to 7.2%. Larger improvement results for dense communication. For example, the combined algorithm uses 43.1% less multiplexing degree than that of the greedy algorithm for all–to–all pattern. This result confirms the result in [33] that it is desirable to use compiled communication for dense communication.

No. of connections	No. of patterns	greedy algorithm	coloring algorithm	AAPC algorithm	combined algorithm	improvement percentage
0 - 100	34	1.2	1.2	1.2	1.2	0.0%
101 - 200	50	5.9	4.9	4.8	4.6	28.3%
200 - 400	54	10.6	9.7	10.0	9.5	11.6%
401 - 800	105	17.7	15.9	16.0	15.5	14.2%
801 - 1200	122	31.7	28.7	28.6	27.6	14.9%
1201 - 1600	0	0	0	0	0	0%
1601 - 2000	15	46.3	42.8	35.1	35.1	31.9%
2001 - 2400	77	55.5	51.5	51.9	50.4	10.1%
2401 - 4031	0	0	0	0	0	0%
4032	43	92	83	64	64	43.8%

Table 5.5: Performance for data distribution patterns

To obtain more realistic results, the performance is also evaluated using the communication patterns for data redistribution and some frequently used communication patterns which occurs in the programs analyzed by the E–SUIF compiler. Table 5.5 shows the performance of the algorithms for data redistribution patterns. The communication patterns are extracted from the communication resulting from the random data redistribution of a 3D array of size $64 \times 64 \times 64$. The random data redistribution is created by randomly generating the source data distribution and the destination data distribution with regard to the number of processors allocated to each dimension and the block size in each dimension. Precautions are taken to make sure that the total processor number is 64 and the block size is not too large so that some processors do not contain any part of the array. The table lists the results for 500 random data redistributions. The first column lists the range of the number of connections in each pattern. The second column lists the number of data redistrictions whose number of connections fell into the range. For example, the second column in the last row indicates that among the 500 random data redistributions, 43 results in 4032 connections. Columns three to six list the multiplexing degree required by the greedy algorithm, the coloring algorithm, the AAPC algorithm and the combined algorithm respectively. The seventh column lists the percentage improvement by the combined algorithm over the greedy algorithm. The result shows that the multiplexing degree required to establish connections resulting from data redistribution is less than that resulting from the random communication patterns. For the data redistribution pattern, the percentage improvement obtained by using the combined algorithm ranges from 10.1% to 31.9%, which is larger than the improvement for the random communication patterns.

Pattern	No. of conn.	greedy	coloring	AAPC	comb	percentage
ring	128	3	2	2	2	50%
nearest neighbor	256	6	4	4	4	50%
hypercube	384	9	7	8	7	28.6%
shuffle–exchange	126	6	4	5	4	50%
all–to–all	4032	92	83	64	64	43.8%

Table 5.6: Performance for frequently used patterns

Table 5.6 shows the performance for some frequently used communication patterns. In the ring and the nearest neighbor patterns, no conflicts arise in the links. However, there are conflicts in the communication switches. The performance gain is higher for these specific patterns when the combined algorithm is used.

5.5 Communication Phase analysis

Armed with the connection scheduling algorithms, the compiler can determine when two communication patterns can be combined so that the underlying network can support both patterns simultaneously and thus, can partition a program into phases such that each phase contains connections that can be supported by the underlying network. This section considers the compiler algorithm to partition a program.

The communication phase analysis is carried out in a recursive manner on the high level SUIF representation of a program, which is similar to an abstract syntax tree. SUIF represents a program in a hierarchical manner. A procedure contains a list of SUIF nodes, where each node can be of different types and can contain sub–lists. Some important SUIF node types include TREE_FOR, TREE_LOOP, TREE_IF, TREE_BLOCK and TREE_INSTR. A TREE_FOR node represents a for–loop structure. It contains four sub–lists, *lb_list* which contains the SUIF to compute the lower bound, *ub_list* which contains the nodes to compute the upper bound, *step_list* which contains the nodes to compute the step, and *body* which contains the loop body. A TREE_LOOP node represents a while–loop structure. It contains two sub–lists, *test* and *body*. A TREE_IF node represents an if–then–else structure. It contains three sub–lists, *header* which is the test part, *then_part* which contains the nodes in the then part, and the *else_part*. A TREE_BLOCK node represents a block of statements, it contains a sub–list *body*. A TREE_INSTR nodes represents a statement.

Given a SUIF representation of a program, which contains a list of nodes, the communication phase analysis algorithm determines the communication phases for each sub–lists in the list and then determines the communication phases of the list. In addition to the annotations for communications, a *composite node*, which contains sub–lists, is associated with two variables, *pattern*, which is the communication pattern that is exposed from the sub–lists, and the *kill_phase*, which has a boolean value, indicating whether its sub–lists contain phases.

The algorithm to analyze communication phases in a program for a node list is shown in Figure 5.15. The algorithm assumes that the multiplexing degree for the system is d. It also uses one of the algorithms discussed in section 5.4, denoted as $multiplexing\_degree(Comm)$, to compute the multiplexing degree required to realize communication pattern $Comm$. Given a node list, the algorithm first recursively examines the sub–lists of all nodes and annotates the nodes with *pattern* and *kill_phase*. This post–order traversal of the SUIF program accumulates the communications in the innermost loops first, and thus can capture the communication locality when it exists and is supported by the underlying network. Figure 5.16 describes the operations for TREE_IF nodes. The algorithms for TREE_IF node computes the phases for the three sub–lists. In the cases when there are phases within the sub–lists and when the network does not have enough capacity to support the combined communication, a phase is created in each of the sub–list to accommodate the

```
Communication_Phase_Analysis(list)
Input: $list$: a list of SUIF nodes
Output: $pattern$: communication pattern exposed out of the list
        $kill\_phase$: whether there are phases within the list

Analyze communication phases for each node in the list.
$c\_pattern = NULL, kill\_phase = 0$
**For each** node $n$ in list in backward order **do**
  **if** ($n$ is annotated with $kill\_phase$) **then**
    Generate a new phase for $c\_pattern$ after $n$.
    $c\_pattern = NULL, kill\_phase = 1$
  **end if**
  **if** ($n$ is annotated with communication pattern $a$) **then**
    $new\_pattern = c\_pattern + a$
    **if** ($multiplexing\_degree(new\_pattern) \leq d$) **then**
      $c\_pattern$ = new_pattern
    **else**
      Generate a new phase for $c\_pattern$ after $n$.
      $c\_pattern = a, kill\_phase = 1$
    **end if**
  **end if**
**end for**
**return** $c\_pattern$ and $kill\_phase$
```

Figure 5.15: Communication phase analysis algorithm

```
Communication_Phase_Analysis for TREE_IF

Analyze the $header$ list.
Analyze the $then\_part$ list.
Analyze the $else\_part$ list.
Let $comb$ = the combined communications from the three sub–lists.
**If** (there are phase changes in the sub–lists) **then**
  Generate a phase in each sub–list for the communication exposed.
  $pattern = NULL, kill\_phase = 1$
**if** ($multiplexing\_degree(comb) > d$) **then**
  Generate a phase in each sub–list for the communication exposed.
  $pattern = NULL, kill\_phase = 1$
**else**
  $pattern = comb, kill\_phase = 0$
**end if**
Annotate the TREE_IF node with $pattern$ and $kill\_phase$.
```

Figure 5.16: Communication phase analysis for TREE_IF nodes

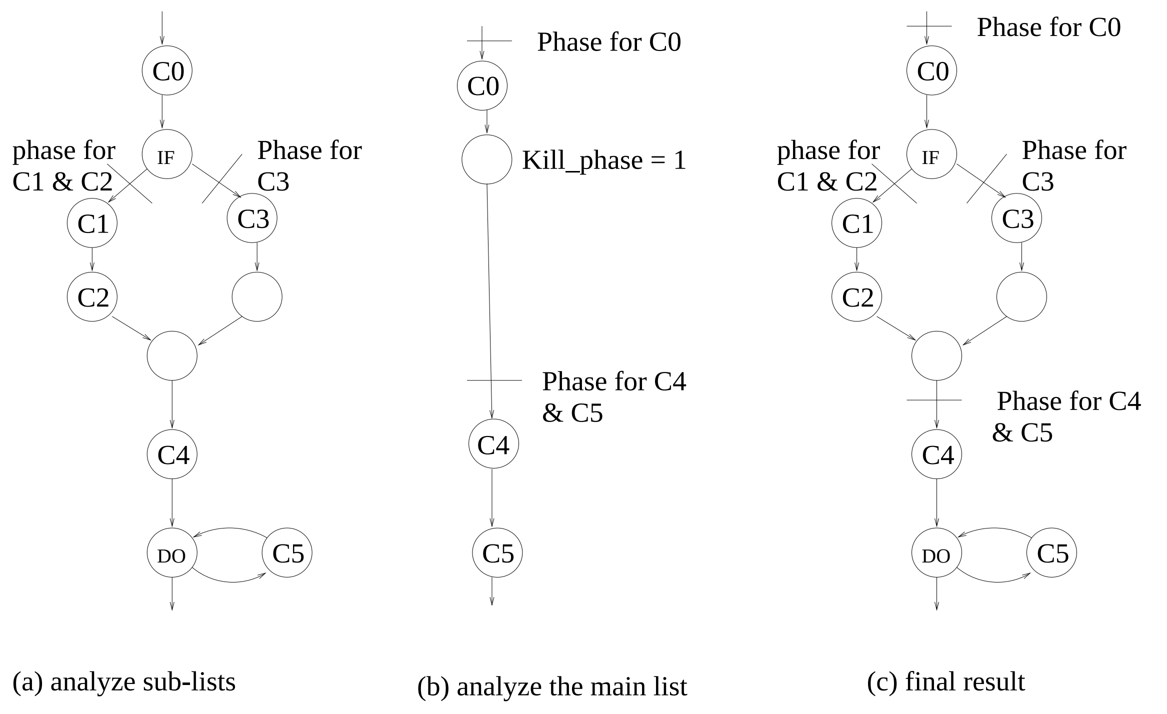


Figure 5.17: An example for communication phase analysis

corresponding communication from that sub–list. Otherwise, the TREE_IF node is annotated with the combined communication indicating the communication requirement of the IF statement. Algorithms for processing other node types are similar. After all sub–lists in all nodes in the list are analyzed, the node list contains a straight line program, whose nodes are annotated with communication, *pattern* and *kill_phase*. The algorithm examines all these annotations in each node from back to front. A variable *c_pattern* is used to maintain all communications currently accumulated. There are two cases when a phase is generated. First, once a *kill_phase* annotation is encountered, which indicates there are phases in the sub–lists, thus, it does not make sense to maintain a phase passing the node since there are phase changes during the execution of the sub–lists, a new phase is created to accommodate the connection requirement after the node. Second, in the cases when adding a new communication pattern into the current (accumulated) pattern exceeds the network capacity, a new communication phase is needed.

Figure 5.17 shows an example for the communication phase analysis. The program in the example contains six communications, $C0$, $C1$, $C2$, $C3$, $C4$, $C5$ and $C6$, an IF structure and a DO structure. The communication phase analysis algorithm first analyzes the sub–lists in the IF and DO structures. Assuming the combination of $C1$ and $C2$ can be supported by the underlying network, while combining communications $C1$, $C2$ and $C3$ exceeds the network capacity, which results in the two phases in the IF branches and the

Prog.	Description	Distrib.
0001	Solution of 2-D Poisson Equation by ADI	(*, block)
0003	2-D Fast Fourier Transform	(*, block)
0004	NAS EP Benchmark - Tabulation of Random Numbers	(*, block)
0008	2-D Convolution	(*, block)
0009	Accept/Reject for Gaussian Random Number Generation	(block)
0011	Spanning Percolation Cluster Generation in 2-D	(*, block)
0013	2-D Potts Model Simulation using Metropolis Heatbath	(*, block)
0014	2-D Binary Phase Quenching of Cahn Hilliard Cook Equation	(*, block)
0022	Gaussian Elimination - NPAC Benchmark	(*, cyclic)
0025	N-Body Force Calculation - NPAC Benchmark	(block, *)
0039	Segmented Bitonic Sort	(block)
0041	Wavelet Image Processing	(*, block)
0053	Hopfield Neural Network	(*, block)

Table 5.7: Benchmarks and their descriptions

Kill_phase is set for the IF header node. Assuming that all communications of $C5$ within the DO loop can be supported by the underlying network, Figure 5.17 (a) shows the results after the sub–lists are analyzed. The algorithm then analyzes the list by considering each node from back to forth, it combines communications $C4$ and $C5$. Since the IF header node is annotated with *kill_phase*. A new phase is generated for communications $C4$ and $C5$ after the IF structure. The algorithm then proceeds to create a phase for communication $C0$. Figure 5.17 (c) shows the final result of the communication phase analysis for this example.

5.5.1 Evalutation of the communication phase analysis algorithm

This section presents the performance evaluation of the E–SUIF compiler for compiled communication. The compiler is evaluated with respect to the analysis time and the runtime performance. The E–SUIF compiler analyzes the communication requirement of a program and partitions the program into phases such that each phase contains a communication pattern that can be realized by a multiplexing degree of d, where d is a parameter. In addition, the compiler also gives channel assignments for connections in each phase. It is assumed that the underlying network is a 8×8 torus.

Programs from the HPF benchmark suite at Syracuse University are used to evaluate the algorithms. The benchmarks and their descriptions are listed in Table 5.7. The table also shows the data distribution of the major arrays in the programs. These distributions are obtained from the original benchmark programs. Table 5.8 breaks down the analysis

benchmarks	size (lines)	overall	logical communication	phase analysis
0001	545	11.33	0.45	8.03
0003	372	24.83	0.50	11.80
0004	599	19.08	0.42	15.02
0008	404	27.08	0.68	13.28
0009	491	46.72	4.45	19.65
0011	439	14.78	0.57	11.37
0013	688	23.08	1.07	17.30
0014	428	15.58	1.03	11.38
0022	496	22.57	0.77	18.35
0025	295	5.77	0.78	3.35
0039	465	16.08	0.38	13.13
0041	579	9.93	0.28	6.62
0053	474	7.39	0.35	4.33

Table 5.8: Communication phase analysis time

time. The table shows the time for overall analysis, the logical communication analysis and the communication phase analysis. The overall analysis includes the time to load and store the program, the time to analyze communication requirement on the virtual processor grid, the time to derive communication requirement on the physical processor grid and the time for communication phase analysis. The communication phase analysis time accounts for a significant portion of the overall analysis time for all the programs. This is because the communication phase operates on large sets of data (communication pattern). However, for medium size programs, such as the benchmarks used, the analysis time is not significant.

Table 5.9 shows the precision of the analysis. It compares the average number of channels and connections per phase obtained from our algorithms with those in actual executions. The number of channels and connections per phase in actual executions is obtained by accumulating the connections within each phase, which is determined by the compiler. When a phase change occurs, the statistics about the number of connections within each phase is collected and the connection scheduling algorithm is invoked to compute the number of channels needed for the connections in that phase. For most programs, the analysis results match the actual program executions, which indicates that approximations are seldom used. For the programs where approximations occur, the channel approximation is better than the connection approximation as shown in benchmark 0022. This is mainly due to the approximation of the communications that are not vectorized. For such communications, if the underlying network can support all connections in a loop, the phase will contain the loop and use the channels for all communications in the loop. However, for the

benchmark	connections per phase			channels per phase		
programs	actual	compiled	percentage	actual	compiled	percentage
0001	564.4	564.4	100%	9.1	9.1	100%
0003	537.6	537.6	100%	8.6	8.6	100%
0004	116.3	116.3	100%	5.5	5.5	100%
0008	562.6	562.6	100%	8.9	8.9	100%
0009	91.2	230.7	39.6%	4.3	6.6	65.1%
0011	126.3	126.3	100%	5.2	5.2	100%
0013	67.3	67.3	100%	3.1	3.1	100%
0014	126.4	126.4	100%	4.0	4.0	100%
0022	13.1	413.2	3%	4.6	8.9	52.7%
0025	80.0	80.0	100%	3.0	3.0	100%
0039	125.7	125.8	99.9%	8.8	8.8	99.9%
0041	556.1	556.1	100%	8.8	8.8	100%
0053	149.2	575.2	25.9%	9.0	9.1	98.9

Table 5.9: Analysis precision

connections, the compiler approximates each individual communication inside the loop with all communications of the loop. Since the number of channels for a communication pattern determines the communication performance, this type of approximation does not hurt the communication performance.

5.6 Chapter summary

This chapter addressed the compiler issues for applying compiled communication. In particular, algorithms for communication analysis were presented which take into consideration common communication optimizations including message vectorization, redundant communication elimination and message scheduling. A demand driven array data flow framework, which improves over previous communication optimization algorithms by reducing the analysis cost and improving the analysis precision, was developed for the communication optimizations. Three off-line connection scheduling algorithms were described that realize a given communication pattern with a minimal multiplexing degree. A communication phase analysis algorithm, which partitions a program into phases such that each phase contains communications that can be supported by the underlying network, was developed. The algorithm also exploits communication locality to reduce the amount of reconfiguration overhead during program execution.

A compiler, called the E–SUIF compiler, implements all the above algorithms and thus, supports compiled communication. The E–SUIF compiler compiles a HPF–like program, analyzes its communication requirement, partitions the program into phases such that each phase contains connections that can be supported by the underlying network, assigns channels for connections in each phase, and outputs a C program with the communication and phase annotations such that when the program is executed, the communications (and phases) in the program can be simulated. All the algorithms were evaluated in the compiler. It was found that the communication optimization algorithms are efficient in terms of the analysis cost and are effective in finding the optimization opportunities. The communication phases analysis algorithm generally captures the program runtime behavior accurately.

In the last three chapters, techniques for the three communication schemes are discussed. Next chapter evaluates the three communication schemes and compares their performance using real application programs.

Chapter 6
Performance comparison

This chapter evaluates the relative performance of the three communication schemes presented in Chapters 3, 4, and 5 using real application programs. Three sets of programs are used in the evaluation. The first set of programs includes three hand–coded parallel programs, where communications are well defined and highly optimized for parallel execution. The second set of programs includes a number of HPF benchmark programs which are tuned for parallel execution. The third set of programs includes a number of programs from SPEC95 which are not optimized for parallel execution.

The performance measurement is the communication time in the unit of time slots. A packet, which contains a number of words, can be transmitted through a lightpath in a time slot. In addition, *normalized time* is also used to compare the performance of the schemes. In normalized time, the best communication time among all schemes is assigned a value of 1.0 and communication times of all schemes are normalized with respect to the best communication time. The normalized time shows the best scheme for each program and how other schemes perform compared to the best scheme. It is assumed that the communication in each pattern is performed in a synchronized manner. That is, the program synchronizes before and after each communication pattern and thus no interleaving of communications and computations is allowed.

Because the E–SUIF compiler does not handle the message passing paradigm, the first set of experiment is carried out manually by extracting the communication patterns in the programs by hand. The programs in the second and third sets are generated automatically by the E–SUIF compiler for the experiment. As discussed in Chapter 5, the E–SUIF compiler first analyzes and optimizes the communications in a program and represents the communications using Section Communication Descriptors (SCDs). It then performs the communication phase analysis and partitions the program into phases and schedules the communication pattern within each phase. Finally, the backend of the E–SUIF compiler generates a library call, *lib_comm*, for each SCD and another library call, *lib_phase* for each phase. The *lib_comm* takes a SCD with all runtime information as parameter. When the program is executed, the *lib_comm* procedure invokes a network simulator which simulates dynamic single–hop communication, dynamic multi–hop communication or compiled communication to obtain the communication time of the communication using one of the three communication schemes. The *lib_phase* is useful only when simulating compiled communication. It accesses to the communication requirement of each phase (that is obtained by the compiler), and performs channel assignment for connections within each phase. Thus, the communication performance of a program is obtained by running the program generated by the E–SUIF compiler.

The experiments use the following system settings.

- Physical network: 8×8 torus.
- Packet size: 4 words.
- Routing algorithm: XY routing between dimensions and Odd–Even shortest–path routing within each dimension.
- Dynamic single–hop communication.
  - Control protocol: Conservative backward reservation protocols (*cset* size is 1). As discussed in Chapter 3, the conservative backward reservation protocol almost has the best performance among all the path reservation protocols.
  - Control packet processing time: 1 time slot.
  - Control packet propagation time: 1 time slot.
  - Maximum control packet retransmission time: 5 time slot.
  - Multiplexing degree: 1, 4, 14, 20.
- Dynamic multi–hop communication.
  - Logical topologies: torus, hypercube, allXY and all–to–all.
  - packet switching time: 1 time slot.
- Compiled communication.
  - Connection scheduling algorithms: combined algorithm for the first set of experiment, AAPC algorithm for the second and third experiments.

6.1 Hand–coded parallel programs

This set of program includes three hand–coded parallel programs, namely *GS*, *TSCF* and *P3M*. The *GS* program uses Gauss–Siedel iterations to solve Laplace equation on a discretized unit square with Dirichlet boundary conditions. It contains a nearest neighbor communication pattern with fairly large message size (64 packets messages). The *TSCF* program simulates the evolution of a self–gravitating system using a self consistent field approach. It contains a hypercube communication pattern with small message size (1 packet message). *P3M* performs particle–particle particle–mesh simulation [84]. This program contains five static communication patterns. Table 6.1 describes the static communication patterns that arise in these programs.

Pattern	Type	Description
GS	shared array ref.	PEs are logically linear array, Each PE communicates with two PEs adjacent to it.
TSCF	explicit send/recv	hypercube pattern
P3M 1	data redistrib.	(:block, :block, :block) $\rightarrow$ (:, :, :block)
P3M 2	data redistrib.	(:, :, :block) $\rightarrow$ (:block, :block, :)
P3M 3	data redistrib.	(:block, :block, :) $\rightarrow$ (:, :, :block)
P3M 4	data redistrib.	(:, :, :block) $\rightarrow$ (:block, :block, :block)
P3M 5	shared array ref.	PEs are logically 3–D array, each PE communicates with 6 PEs surrounding it

Table 6.1: Communication pattern description.

Table 6.2 shows the communication time for these patterns in one main loop step in the programs. Table 6.3 shows the normalized time where the best communication time is normalized to 1.0. In this experiment, it is assumed that there is sufficient multiplexing degree to support all the patterns in compiled communication. Thus, each phase contains one communication pattern and no network reconfiguration is required to within each pattern. For dynamic single–hop communication, the communication time for fixed multiplexing degrees of 1, 4, 14 and 20 is evaluated. For dynamic multi–hop communication, the logical torus, hypercube, allXY and all–to–all topologies are considered. The following observations can be made from the results in Table 6.3.

- Compiled communication out–performs dynamic single–hop communication in all cases. The average communication time for dynamic single–hop communication is 4.5 to 8.0 times greater than that for compiled communication, depending on the multiplexing degree used in dynamic single–hop communication. Larger performance gains are observed for communications with small message sizes (e.g., the $TSCF$ pattern) and dense communication (e.g., the $P3M$ 2 pattern). Large multiplexing degree does not always improve the communication performance for dynamic single–hop communication. For example, a multiplexing degree of 1 results in the best performance (for dynamic single–hop communication) for the pattern in GS while a degree of 14 has the best performance for the $P3M$ 5 pattern.

- Compiled communication out–performs dynamic multi–hop communication in all cases except for the $TSCF$ program where dynamic multi–hop communication has better communication time when using the logical hypercube topology. The reason is that the $TSCF$ program only contains hypercube communication with message size equal to 1. Multi–hop communication achieves good communication performance when communication patterns in a program matches the logical topology. However, on average, the communication time for multi–hop communication is 3.0 to 7.6 times larger than the communication time for compiled communication, depending on the logical topology used.

- Compiled communication achieves an average normalized time of 1.1 for all the communication patterns, which indicates that compiled communication almost delivers optimal communication performance.

- Comparing dynamic multi–hop communication with dynamic single–hop communication, multi–hop communication has better performance when the message size is small (e.g. $TSCF$, $P3M$ 5), and when the communication requires dense connections (e.g. $P3M$ 2, 3), while single–hop communication is better when the message size is large (e.g. GS).

Pattern		GS	TSCF	P3M 1	P3M 2,3	P3M 4	P3M 5
Compiled comm.		131	19	831	382	457	40
Multihop comm.	torus	404	30	3366	1656	1632	127
	hypercube	792	13	3371	1338	1499	74
	allXY	990	17	3157	1058	960	121
	alltoall	4159	70	1326	749	1326	276
Single–hop comm	$d = 1$	209	215	3194	6655	2091	378
	$d = 4$	296	118	2029	2998	1302	213
	$d = 14$	924	107	1713	2171	1508	196
	$d = 20$	1296	108	1702	2096	1314	231

Table 6.2: Communication time (timeslots) for the hand–coded programs

Pattern		GS	TSCF	P3M 1	P3M 2,3	P3M 4	P3M5	Average
Compiled comm.		1.0	1.5	1.0	1.0	1.0	1.0	1.1
Multihop comm.	torus	3.1	2.3	4.1	4.3	3.6	3.2	3.4
	hypercube	6.0	1.0	4.1	3.5	3.3	1.9	3.3
	allXY	7.6	1.3	3.8	2.8	2.1	3.0	3.4
	alltoall	31.7	5.4	1.6	2.0	2.9	6.9	8.4
Single–hop comm	$d = 1$	1.6	16.5	3.9	17.4	4.6	9.5	8.9
	$d = 4$	2.3	9.1	2.4	7.8	2.8	5.4	5.0
	$d = 14$	7.1	8.2	2.1	5.7	3.3	4.9	5.2
	$d = 20$	9.9	8.2	2.0	5.5	2.9	5.8	5.7

Table 6.3: Normalized communication time for the hand–coded programs

In this study, two types of communication patterns are observed in a well designed parallel program, fine grain communications resulted from shared array references and coarse grain communications resulted from data redistributions. The fine grain communication causes sparse connections with small message sizes, while the coarse grain communication results in dense connections with large message size. For a communication system to efficiently support the fine grain communication, the system must have small latency. Optical single–hop networks that use dynamic path reservation algorithms have a large startup overhead, thus cannot support this type of communication efficiently. As shown in our simulation results, compiled communication where the startup overhead is eliminated and dynamic multi–hop communication perform this type of communications efficiently. For the coarse grain communication, the control overhead in the dynamic communications is not significant. However, dense communication results in a large number of conflicts in the system (path reservation in dynamic single–hop communication and packet routing in dynamic multi–hop communication), and the dynamic control systems are not able to resolve these conflicts efficiently. By using an off–line connection scheduling algorithm, compiled communication handles this type of communications efficiently. The performance study confirms the conclusion in [33] that static management of the dense communication patterns results in large performance gains.

6.2 HPF parallel benchmarks

This set of programs is from the Syracuse University HPF benchmark suite. The benchmarks and their descriptions are listed in table 5.7 in Section 5.5.1 . The benchmarks include many different types of applications, however, all of the programs contain only regular computations.

The major difference between this experiment and the first experiment is that, in this experiment, compiled communication is applied to the whole program instead of each individual communication pattern. Assuming a multiplexing degree of 10, the compiler tries to aggregate as many communications as possible into a phase as opposed to the first experiment where compiled communication is assumed to have an infinite number of virtual channels to handle each individual communication pattern in the programs. In addition, this set of programs contains communication patterns about which the compiler cannot obtain precise information. Two factors may degrade the performance of compiled communication. First, compiler approximations may result in the waste of bandwidth for establishing connections that are not used. Second, aggregating more communications in a phase reduces the number of network reconfigurations, but may result in larger communication time since larger multiplexing degree is needed for more communications. This experiment aims at studying the performance of compiled communication under these limitations.

Table 6.4 shows the communication time of the programs using different communication schemes. Table 6.5 shows the normalized time. Even with the limitations discussed earlier, compiled communication in general out–performs dynamic communications to a large degree. The benefits of managing channels at compile time and eliminating the runtime path reservation overhead over–weights the bandwidth losses through the imprecision of compiler analysis. The average normalized time for compiled communication is 1.1 which indicates that compiled communication almost delivers the best communication performance for this set of programs. However, performance degradation in compiled communication due to the conservative approximation in compiler analysis is observed in some of the programs. For example, compiler over–estimating the communication requirement is found in benchmarks 0009 and 0022. Note that the overall communication time for the programs in Table 6.4 may not show this, because each program contains many communication patterns and the pattern that is approximated may not dominate the overall communication time. The performance loss due to aggregating communications, which results in larger multiplexing degree, is observed in benchmark 0025. Nonetheless, the overall trend of this experiment is very similar to that in the first experiment.

6.3 Programs from SPEC95

Four programs, ARTDIF (from HYDRO2D), TOMCATV, SWIM and ERHS (from APPLU) are used in this experiment. These programs are also used in Section 5.2.4, where the descriptions of these programs can be found, to evaluate performance of the communication analyzer in the E–SUIF compiler.

benchmarks		0001	0003	0004	0008	0009	0011
Compiled comm.		45,624	752	1,368	2,256	2,394	105,252
Multihop comm.	torus	197,760	3,296	1,776	9,888	3,108	158,594
	hypercube	159,840	2,664	1,032	7,992	1,806	147,496
	allXY	125,280	2,088	1,704	6,439	2,982	265,636
	alltoall	87,960	1,466	4,944	4,398	8,652	1,027,818
Single–hop comm	$d = 1$	888,240	14,804	1,920	44,412	3,360	141,052
	$d = 4$	357,600	5,960	2,208	17,880	3,864	181,506
	$d = 14$	267,360	4,456	3,504	13,368	6,132	372,678
	$d = 20$	273,360	4,556	4,224	13,668	7,392	484,374

benchmarks		0013	0014	0022	0025	0039	0041
Compiled comm.		166,280	63,400	3,244,819	29,854	68,704	1,504
Multihop comm.	torus	257,980	129,800	6,382,683	25,470	106,525	6,592
	hypercube	363,340	200,600	9,509,070	58,661	132,348	5,328
	allXY	748,220	379,200	5,922,920	63,264	135,353	4,176
	alltoall	3,368,600	1,679,200	6,379,275	214,343	393,166	2,932
Single–hop comm	$d = 1$	154,080	71,200	6,844,054	23,440	115,488	29,608
	$d = 4$	256,240	125,200	6,402,631	31,221	136,390	11,920
	$d = 14$	779,920	391,200	6,516,485	61,712	214,042	8,912
	$d = 20$	1,086,160	550,800	6,925,278	81,958	261,832	9,112

Table 6.4: Communication time for the HPF benchmarks.

benchmarks		0001	0003	0004	0008	0009	0011
Compiled comm.		1.0	1.0	1.3	1.0	1.3	1.0
Multihop comm.	torus	4.3	4.4	1.7	4.4	1.7	1.5
	hypercube	3.5	3.5	1.0	3.5	1.0	1.4
	allXY	2.7	2.3	1.7	2.9	1.7	2.5
	alltoall	1.9	1.9	4.8	2.0	4.8	9.8
Single–hop comm	$d = 1$	19.3	19.7	1.9	19.7	1.9	1.3
	$d = 4$	7.8	7.9	2.1	7.9	2.1	1.7
	$d = 14$	5.8	5.9	3.4	5.9	3.4	3.5
	$d = 20$	5.9	6.0	4.1	6.0	4.1	4.6

benchmarks		0013	0014	0022	0025	0039	0041	average
Compiled comm.		1.1	1.0	1.0	1.3	1.0	1.0	1.1
Multihop comm.	torus	1.7	2.1	2.0	1.1	1.6	4.4	2.6
	hypercube	2.4	3.2	2.9	2.5	1.9	3.5	2.5
	allXY	4.9	6.0	1.8	2.7	2.0	2.8	2.8
	alltoall	21.9	26.7	2.0	9.1	5.7	1.9	7.6
Single–hop comm	$d = 1$	1.0	1.1	2.1	1.0	1.7	19.7	7.5
	$d = 4$	1.7	1.9	2.0	1.3	2.0	7.9	3.9
	$d = 14$	5.1	6.2	2.0	2.7	3.1	5.9	4.4
	$d = 20$	7.1	8.7	2.1	3.6	3.8	6.1	5.2

Table 6.5: Normalized time for the HPF benchmarks.

Pattern		ARTDIF	TOMCATV	SWIM	ERHS
Compiled comm.		1,224	15,480	2,708	6,689
Multihop comm.	torus	2,724	34,260	1,378	4,380
	hypercube	4,338	57,240	2,309	6,482
	allXY	8,583	108,900	4,409	15,117
	alltoall	38,772	491,460	19,169	68,800
Single–hop comm	$d = 1$	666	8,280	669	1,148
	$d = 4$	2,478	31,260	1,574	4,382
	$d = 14$	8,538	108,060	4,511	15,166
	$d = 20$	12,168	154,140	6,343	21,614

Table 6.6: Communication time for SPEC95 benchmark programs.

Pattern		ARTDIF	TOMCATV	SWIM	ERHS	average
Compiled comm.		1.8	1.9	4.0	5.8	3.3
Multihop comm.	torus	4.1	4.1	2.1	3.8	3.5
	hypercube	6.5	6.9	3.5	5.6	5.6
	allXY	12.9	13.1	6.6	13.1	11.4
	alltoall	58.2	59.2	28.7	59.9	51.5
Single–hop comm	$d = 1$	1.0	1.0	1.0	1.0	1.0
	$d = 4$	3.7	3.8	2.3	3.8	3.4
	$d = 14$	12.8	13.0	6.7	13.2	11.4
	$d = 20$	18.2	18.6	9.4	18.8	16.3

Table 6.7: Normalized communication time for SPEC95 benchmark programs.

Table 6.6 shows the communication performance of the programs. Table 6.7 shows the normalized time. The test inputs are used as the inputs to these program, which determine the problem size. To reduce the simulation time, the main iteration numbers in programs ARTDIF, SWIM and ERHS are reduced to one. All programs ARTDIF, SWIM, TOMCATV and ERHS only contains simple nearest neighbor communication patterns. Compiled communication performs worse than dynamic single–hop communication with a multiplexing degree of one because it aggregates communications and uses larger multiplexing degree than needed. Hence, it is desirable to develop more advanced communication phase analysis techniques that can use different multiplexing degrees for different parts of a program to achieve best performance. However, considering all the programs evaluated, compiled communication out–performs other schemes to a large degree as shown in Table 6.8.

Comm. schemes	Comp. comm.	Multi–hop				Single–hop			
		torus	hype.	allXY	alltoall	1	4	14	20
Norm. time	1.5	3.0	3.3	4.6	16.1	6.6	4.0	5.9	7.4

Table 6.8: Average normalized communication time for each scheme.

6.4 Chapter summary

This chapter studied the communication performance for the three communication mechanisms, dynamic single–hop communication, dynamic multi–hop communication and compiled communication using three sets of programs. The following conclusions were drawn from the study.

- Compiled communication out–performs dynamic communications to a large degree for applications with regular computations.

- The performance of compiled communication can be further improved by incorporating more advanced communication phase analysis techniques that allow different multiplexing degrees in different parts of a program.

- The major disadvantage of dynamic communications is that they cannot adapt to different communication requirements. Thus, they support some communication patterns efficiently while they are inefficient for other communication patterns. Compiled communication efficiently supports all types of communication patterns that can be determined at compile time.

- Comparing dynamic multi–hop communication and dynamic single–hop communication, dynamic multi–hop communication achieves better performance when the message size is small and when the communication is dense, while dynamic single–hop communication is better when the message size is large. This result matches the results in Section 4.5 where dynamic single–hop communication is compared with dynamic multi–hop communication using randomly generated communication patterns.

Chapter 7
Conclusion

While optical interconnection networks have the potential to provide very large bandwidth, network control, which is performed in the electronic domain due to the lack of suitable photonic logic devices, has become the communication bottleneck in such networks. In order to design efficient optical networks where end users can utilize the large bandwidth, efficient network control mechanisms must be developed to reduce the control overheads. This thesis addresses the network control bottleneck problem in optical networks by considering three communication schemes, *dynamic single–hop communication*, *dynamic multi–hop communication* and *compiled communication*. In addition to developing techniques to improve communication performance in each scheme, this thesis also compares the communication performance of the three schemes and identifies the advantages and the limitations of each scheme. In the following sections, the thesis contributions are summarized and directions for future research are identified.

7.1 Thesis contributions

This thesis makes contributions in the *design of control mechanisms* for time–multiplexed optical interconnection networks. The contributions are in two areas: optical interconnection networks and compiler analysis techniques. In the optical interconnection networks area, this thesis introduces efficient control schemes for dynamic single–hop communication and dynamic multi–hop communication. This thesis also proposes and validates the idea of applying the compiled communication technique to optical TDM networks. In the compiler area, this thesis addresses all the issues needed to apply the compiled communication paradigm to optical interconnection networks, including communication optimization, communication analysis, connection scheduling and communication phase analysis. The main contributions of the thesis are detailed as follows.

- **Dynamic single–hop communication**. Two sets of efficient path reservation algorithms, *forward path reservation protocols* and *backward path reservation protocols*, are designed. Variants of the protocols, including holding/dropping and aggressive/conservative schemes, are considered. The performance of the protocols and the impact of system parameters on these protocols are evaluated. Forward path reservation protocols extend traditional path reservation schemes for electronic networks and are simpler compared to backward path reservation protocols. However, these schemes suffer from either the over–locking problem for aggressive schemes or the low successful reservation rate for conservative schemes. Backward path reservation protocols overcome these problems by probing the network state before reserving channels. Performance study has established that in optical time–division multiplexing networks, backward path reservation protocols, though more complex than forward path reservation protocols, result in better communication performance when the corresponding system and protocol parameters are the same. It is also found that while some system or protocol parameters, such as the holding time for holding schemes, do not have a significant impact on the performance of the protocols, other parameters, such as the aggressiveness of a protocol and the speed of the control network, affect the performance drastically. Similar techniques can be extended for the path reservation in WDM wide area networks [78, 87].

- **Dynamic multi–hop communication**. Schemes for realizing four logical topologies, torus, hypercube, allXY and all–to–all, on top of the physical torus topologies are considered. Optimal and near optimal routing and channel assignment (RCA) schemes for realizing hypercube on array, ring, mesh and torus topologies are developed. An analytical model for analyzing the maximum throughput and the average packet delay is developed and verified via simulation. This model is used to study the performance of dynamic multi–hop communication using the four logical topologies. It is found that in terms of the maximum throughput, the logical all–to–all topology is the best while the logical torus topology has the lowest performance. In terms of the average packet delay, the logical torus topology achieves best results only when the router is fast and the network is under light load, while the logical all–to–all topology is best only when the router is slow and the network is almost saturated. In all other cases, logical hypercube and allXY topologies out–perform logical torus and all–to–all topologies. In addition, the impact of system parameters, such as the packet switching time, on these topologies are studied. In general, the performance of the logical topologies with low connectivity, such as the torus and hypercube topologies, are more sensitive to the network load and the router speed while the logical topologies with more connectivity, such as the all–to–all and allXY topologies, are more sensitive to network size. Some of the techniques developed for multi–hop communication in optical TDM networks can be applied to other areas. The optimal scheme to realize hypercube on mesh–like topologies can be used to efficiently perform communications in algorithms that contain hypercube communication patterns [50]. The modeling technique can be extended to the modeling of WDM networks or electronic networks that perform multi–hop communication.

- **Compiled communication**. This thesis considers all the issues necessary to apply compiled communication to optical TDM networks, including communication optimization, communication analysis, connection scheduling and communication phase analysis.

  - *Communication optimization and communication analysis.* A communication descriptor called *Section Communication Descriptor* (SCD) that describes communications on virtual processor grids is developed. A communication analyzer which performs a number of communication optimizations, including message vectorization, redundant communication elimination and message scheduling, is presented. All the optimizations use a demand driven global array data flow analysis framework. This framework improves previous data flow analysis algorithms for communication optimizations by reducing analysis cost and increasing analysis precision. Algorithms are developed to derive communications on physical processors from SCDs. These algorithms address the problem of effective approximations in the cases when the information in a SCD is insufficient for deriving precise communication on physical processors. The communication optimization technique is general and can be implemented in a compiler that compiles HPF–like programs for distributed memory machines. The communication analysis technique can be used by a compiler that requires the knowledge of the communication requirement of a program on physical processors.

  - *Connection scheduling.* A number of heuristic connection scheduling algorithms are developed to schedule connections on torus topologies. Some of the algorithms can also be applied to other topologies.

  - *Communication phase analysis.* A communication phase analysis algorithm is designed to partition a program into phases such that each phase contains communications that can be supported by the underlying network, while capturing the communication locality in the program to reduce the reconfiguration overheads. This algorithm can also be applied to compiled communication on electronic networks.

- **Communication performance comparison**. A number of benchmarks and real application programs, including hand–coded parallel programs, HPF kernel benchmarks and programs from SPEC95, are used to compare the communication performance of the three communication schemes. The relative strengths and weaknesses of the three schemes are evaluated. The study establishes that even with the limitations of compiler analysis, compiler communication generally out–performs dynamic communications. It delivers high communication performance for all types of communication patterns that are known at compile time. The dynamic single–hop communication and dynamic multi–hop communication both suffers from the inability to adapt to the communication requirement. Given a fixed system setting, they provide good performance for some communication patterns while fail to achieve high performance for other communication patterns. Comparing these two communication schemes, multi–hop communication has the advantage when the message size is small and when the communication requires dense connections, while single–hop communication has the advantage when the message size is large.

7.2 Future research

The research of this thesis can be extended in various ways. Some of the algorithms can be improved. Additional work may either extend the applicability of the techniques or improve the techniques. Following are a number of future research directions that are related to this thesis.

- **Improving backward path reservation algorithms**. In the backward reservation, once a channel is reserved, the reservation fails only when the network state changes. Due to the distributed manner of collecting channel states and reserving channels in backward path reservation algorithms, the information for channels on links close to the source node is not as accurate as the information for channels on links close to the destination node. This problem can be severe when the network size is large. Two possible solutions to this problem are as follows. First, a more efficient control network can be used to route control messages. For example, a Multistage Interconnection Network (MIN) with multi–cast capability can be used to route control messages so that control messages can reach all nodes along the path at the same time. This allows a protocol to collect the channel usage information more efficiently and increases the chance of successful reservation. Second, assuming that the control network has the same topology as the data network, the backward path reservation protocols can selectively collect the channel usage information. The idea behind this improvement is that wrong information may be worse than no information.

- **Path reservation with adaptive routing**. In the thesis, path reservation algorithms assume a deterministic routing. Preliminary research on extending the path reservation protocols with adaptive routing was carried out. The preliminary results show that using current path reservation protocols (both forward and backward reservations), the adaptive routing yields lower maximum throughput on the physical torus topology for uniform communication traffics. Further research is needed to explain this phenomenon and to design path reservation protocols that take advantage of adaptive routing.

- **Topologies for multi–hop communication**. In this thesis, four logical topologies, torus, hypercube, allXY and alltoall, on top of the physical torus topologies are considered. There are two ways to extend this work. First, a different physical topology can be considered. For instance, it would be interesting to consider efficiently realizing regular topologies, such as mesh, torus, on top of an irregular topology. Second, there are logical topologies other than the four logical topologies considered that can achieve good communication performance. Examples include the tree and the shuffle–exchange topologies.

- **Interprocedural communication optimization**. The communication analyzer in the thesis performs a number of communication optimizations, including message vectorization, global redundant communication elimination and global message scheduling, using intraprocedural array data flow analysis. By incorporating the interprocedural array data flow analysis, more optimization opportunities can be uncovered. The intraprocedural array data flow analysis framework uses interval analysis. It can naturally be extended to interprocedual analysis by treating a procedure as an interval. However, many details, such as array reshaping at subroutine boundaries and its impact on communications, must be considered in order for the interprocedural analysis to work.

- **Improving communication phase analysis**. The communication phase analysis algorithm in the thesis follows simple heuristics, it considers the control structures in a program using post–order traversal. This enables the algorithm to consider communications in innermost loops first, aggregate the communications out of loops to reduce the reconfiguration overhead and capture the communication locality. However, while the algorithm is simple to implement, the phases it generates are not optimal in the sense that there may exist other program partitioning schemes that result in less phases in a program. More advanced communication phase analysis algorithms based on better communication model [66] may be developed by using a general control flow graph for program representation and by considering the communication requirement of the whole procedure when generating phases.

- **Compact communication descriptor**. The communication descriptor in the compiler that describes communication patterns on physical processors is a flat structure. It contains all pairs of source and destination nodes. This descriptor is both large and hard to manipulate. More compact communication descriptor is desirable for the compiler. The challenge however, is that the descriptor must both be compact and easy to use by the analysis algorithms.

- **Irregular communication patterns**

  Many scientific codes contain irregular communication patterns that can only be determined at runtime. This thesis has restricted the compiled communication technique to be applied to the programs that contain only regular computations. This restriction can be relaxed by using a strategy similar to the Chaos runtime library[74]. This library performs an inspector phase that calculates the runtime schedule once for many executions of the communication pattern. Similarly the connection scheduling algorithms can gather communication information at runtime and assign channels to all connections within the next looping structure to be used for subsequent iterations.

7.3 Impact of this research

This thesis establishes that the compiled communication technique is more efficient than both dynamic single–hop communication and dynamic multi–hop communication. The compiler algorithms that enable the application of compiled communication on optical TDM networks, though can be further improved, are available in this thesis. Although the compiled communication technique can only apply to the communication patterns that are known at compile time, mechanisms that allow the compiler to manage network resources so that compiled communication can be supported must be incorporated in future optical TDM networks for multiprocessor systems to achieve high performance. Dynamic communication schemes must be used to handle general communication in an optical TDM network.

Dynamic single–hop communication incurs large startup overhead and is thus inefficient for small messages which occur frequently in parallel applications. Dynamic multi–hop communication is efficient for small messages, however, it places electronic processing in the critical path of data transmission and cannot fully utilize the large bandwidth in optical links when the optical data transmission speed is significantly faster than the electronic processing speed. Hence, both schemes have their own advantages and the better choice between these two schemes depends on the application programs and the advances in optical networking technology.

This thesis develops techniques for efficient communication in optical TDM networks. Many techniques developed can be applied to other areas. The path reservation algorithms for dynamic single–hop communication can be extended for WDM wide area networks. The efficient routing and channel assignment algorithms for hypercube communication pattern can be used to efficiently perform communications in algorithms that contain hypercube communication patterns. The modeling technique for multi–hop communication in optical TDM networks can be extended for WDM networks and electronic networks with multi–hop communication. The communication optimization technique based on a demand driven data flow analysis technique can be incorporated in a compiler that compiles a HPF–like language for distributed memory machines. The communication analysis technique can be used by compilers that perform architectural dependent communication optimizations, or compiled communication on electronic networks.

Bibliography

Bibliography

[1] A.S. Acampora and M.J. Karol, "An Overview of Lightwave Packet Network." *IEEE Network Mag.* 3(1), pages 29-41, 1989.

[2] S. P. Amarasinghe and M. S. Lam "Communication Optimization and Code Generation for Distributed Memory Machine." In *Proceedings ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, June 1993.

[3] S. P. Amarasinghe, J. M. Anderson, M. S. Lam and C. W. Tseng, "The SUIF Compiler for Scalable Parallel Machines." *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, February, 1995.

[4] H.R. As, "Media Access Techniques: the Evolution towards Terabit/s LANs and MANs." *Computer Networks and ISDN Systems*, 26(1994) 603–656.

[5] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. "The PARADIGM Compiler for Distributed-Memory Multicomputers." in *IEEE Computer*, Vol. 28, No. 10, pages 37-47, October 1995.

[6] J.A. Bannister, L. Fratta and M. Gerla "Topological Design of the Wavelength–Division Optical Network." *IEEE INFOCOM'90*, pages 1005—1013, 1990.

[7] R.A. Barry and P.A. Humblet. "Models of Blocking Probability in All–optical Networks with and without Wavelength Changers." In *Proceeding of IEEE Infocom*, pages 402-412, April 1995.

[8] B. Beauquier, J. Bermond, L. Gargano, P. Hell, S. Perennes and U. Vaccaro "Graph Problems Arraying from Wavelength–Routing in All–Optical Networks." *Workshop on Optics and Computer Science*, 1997.

[9] C. A. Brackett, "Dense wavelength division multiplexing networks: Principles and applications," *IEEE Journal on Selected Areas of Communications*, Vol. 8, pp. 948-964, Aug. 1990.

[10] J. Brassil, A. K. Choudhury and N.F. Maxemchuk, "The Manhattan Street Network: A High Performance, Highly Reliable Metropolitan Area Network," *Computer Networks and ISDN Systems*, 26(6-8), pages 841-858, 1994.

[11] M. Bromley, S. Heller, T. McNerney and G. L. Steele, Jr. "Fortran at Ten Gigaflops: the Connection Machine Convolution Compiler'." In *Proc. of SIGPLAN'91 Conf. on Programming Language Design and Implementation.* June, 1991.

[12] D. Callahan and K. Kennedy "Analysis of Interprocedural Side Effects in a Parallel Programming Environment." *Journal of Parallel and Distributed Computing*, 5:517-550, 1988.

[13] F. Cappelllo and C. Germain. "Toward high communication performance through compiled communications on a circuit switched interconnection network." In *Proceedings of the Int'l Symp. on High Performance Computer Architecture*, pages 44-53, Jan. 1995.

[14] S. Chakrabarti, M. Gupta and J. Choi "Global Communication Analysis and Optimization." *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation* (PLDI), Pages 68 — 78, Philadelphia, PA, May, 1996.

[15] B. Chapman, P. Mehrotra and H. Zima "Programming in Vienna Fortran." *Scientific Programming*, 1:31–51, Fall 1992.

[16] S. Chatterjee, J. R. Gilbert, R. Schreiber and S. Teng "Automatic Array Alignment in Data–Parlllel Programs." *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, Jan. 1993.

[17] S. Chatterjee, J. Gilbert, F. J. E. Long, R. Schreiber and S. Teng "Generating local addresses and communication sets for data–parallel programs." In *Proc. of PPoPP*, pages 149–158, San Diego, CA, May 1993.

[18] C. Chen and s. Banerjee, "A New Model for Optimal Routing and Wavelength Assignment in Wavelength Division Multiplexed Optical Networks," *Proc. IEEE Infocom'96*, pages 164–171, 1996.

[19] I. Chlamtac, A. Ganz and G. Karmi. "Lightpath Communications: An Approach to High Bandwidth Optical WAN's" *IEEE Trans. on Communications*, Vol. 40, No. 7, July 1992.

[20] I.Chlamtac, A. Ganz and G. Karmi "Lightnets: Topologies for High–Speed Optical Networks." *Journal of Lightwave Technology*, Vol. 11, No. 5/6, pages 951—961, May/June 1993.

[21] W. Dally and C. Seitz, "Deadlock–Free Message Routing in Multiprocessor Interconnection Networks." *IEEE trans. on Computers*, Vol. C–36, No. 5, May 1987.

[22] P. Dowd, K. Bogineni and K. Ali, "Hierarchical Scalable Photonic Architectures for High-Performance Processor Interconnection", *IEEE Trans. on Computers*, vol. 42, no. 9, pp. 1105-1120, 1993.

[23] A. Ganz and Y. Gao, "A Time-Wavelength assignment algorithm for WDM Start Networks", *Proc. of IEEE INFOCOM*, 1992.

[24] C. Gong, R. Gupta and R. Melhem. "Compilation Techniques for Optimizing Communication on Distributed-Memory System". *International conference on Parallel Processing.* Vol. II, pages 39-46, August 1993.

[25] T. Gross. "Communication in iWarp Systems." In *Proceedings Supercomputing*'89, pages 436–445, ACM/IEEE, Nov. 1989.

[26] T. Gross, A. Hasegawa, S. Hinrichs, D. O'Hallaron, and T. Stricker "Communication Styles for Parallel Systems." *IEEE Computer*, vol.27, no. 12, December, 1994, pp. 34-44.

[27] T. Gross, D. O'Hallaron, and J. Subhlok "Task parallelism in a High Performance Fortran framework." *IEEE Parallel & Distributed Technology*, vol 2, no 2, 1994, pp 16-26.

[28] M. Gupta and P. Banerjee. "A Methodology for High–Level Synthesis of Communication on Multicomputers." In *International Conference on Supercomputing*, Pages 357–367, 1992.

[29] M. Gupta and P. Banerjee. "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers." *IEEE Trans. on Parallel and Distributed Systems*, 3(2)179-193, 1992.

[30] M. Gupta and E. Schonberg "A Framework for Exploiting Data Availability to Optimize Communication." In *6th International Workshop on Languages and Compilers for Parallel Computing*, LNCS 768, pp 216-233, August 1993.

[31] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, K.Y. Wang, D. Shields, W.M. Ching and T. Ngo. "An HPF compiler for the IBM SP2." In *proc. Supercomputing'95*, San Diego, CA, Dec. 1995.

[32] M. Gupta, E. Schonberg and H. Srinivasan "A Unified Framework for Optimizing Communication in Data-parallel Programs." In *IEEE trans. on Parallel and Distributed Systems*, Vol. 7, No. 7, pages 689-704, July 1996.

[33] S. Hinrichs, C. Kosak, D.R. O'Hallaron, T. Stricker and R. Take. "An Architecture for Optimal All–to–All Personalized Communication." In *6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 310-319, June 1994.

[34] S. Hinrichs. "Compiler Directed Architecture–Dependent Communication Optimization." Ph.D dissertation, School of Computer Science, Carnegie Mellon University, 1995.

[35] S. Hinrichs "Simplifying Connection–Based Communication." *IEEE Parallel and Distributed Technology*, 3(1)25–36, Spring 1995.

[36] H. Scott Hinton, "Photonic Switching Using Directional Couplers", *IEEE Communication Magazine*, Vol 25, no 5, pp 16-26, 1987.

[37] S. Hiranandani, K. Kennedy and C. Tseng "Compiling Fortran D for MIMD Distributed–memory Machines." *Communications of the ACM*, 35(8):66-80, August 1992.

[38] T. Horie and K. Hayashi. "All–to–All Personalized Communication on a wrap–around Mesh." In *Proceedings of CAP Workshop*, November, 1991.

[39] High Performance Fortran Forum. *High Performance Fortran Language Specification Version 1.0.*, May 1993.

[40] T. Ikegami "WDM Devices, State of the Art." *Photonic Networks*, Springer, pages 79–90, 1997.

[41] K. Kennedy and N. Nedeljkovic "Combining dependence and data-flow analyses to optimize communication." In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[42] K. Knobe, J.D. Lukas and G.L. Steele, Jr. "Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines." *Journal of Parallel and Distributed Computing*, 8:102-118, 1990.

[43] C. Koelbel "Compiling Programs for Nonshared Memory Machines." Ph.D thesis, Purdue University, August 1990.

[44] C. Koelbel and P. Mehrotra "Compiling global name–space parallel loops for distributed execution." *IEEE Trans. on Parallel and Distributed Systems*, 2(4):440-451, Oct. 1991.

[45] M. Kovacevic, M. Gerla and J.A. Bannister, "On the performance of shared–channel multihop lightwave networks," *Proceedings IEEE INFOCOM'95*, Boston, MA, pages 544–551, April 1995.

[46] M. Kumar. "Unique Design Concepts in GF11 and Their Impact on Performance". *IBM Journal of Research and Development.* Vol. 36 No. 6, November 1992.

[47] J. P. Labourdette and A. S. Acampora "Logically Rearrangeable Multihop Lightwave Networks." *IEEE Trans. on Communications*, Vol. 39, No. 8, pages 1223—1230, August 1991.

[48] D. Lahaut and C. Germain, "Static Communications in Parallel Scientific Programs." In *Parallel Architecture & Languages*, Europe, pages 262–274, Athen, Greece, July 1994.

[49] S. Lee, A. D. Oh and H.A. Choi "Hypercube Interconnection in TWDM Optical Passive Star Networks", *Proc. of the 2nd International Conference on Massively Parallel Processing Using Optical Interconnections.* San Antonio, Oct. 1995.

[50] F. Leighton, *Introduction to parallel algorithms and architecture: arrays, trees, hypercubes.* Morgan Kaufmann, 1992.

[51] J. Li and M. Chen. "Compiling Communication –efficient Programs for Massive Parallel Machines." *IEEE Trans. on Parallel and Distributed Systems*, 2(3):361-376, July 1991.

[52] J. Li and M. Chen "The Data Alignment Phase in Compiing Programs for Distributed Memory Machines." *Journal of Parallel and Distributed Computing*, 13(2):213–221, October 1991.

[53] R. Melhem, "Time–Multiplexing Optical Interconnection Network; Why Does it Pay Off?" In *Proceedings of the 1995 ICPP workshop on Challenges for Parallel Processing*, pages 30–35, August 1995.

[54] "The Message Passing Interface Forum". *Draft Document for a Standard Message Passing Interface*, November 1993.

[55] B. Mukherjee, "WDM–based local lightwave networks — Part I: Single–hop systems," *IEEE Network Magazine*, vol. 6, no. 3, pp. 12–27, May 1992.

[56] B. Mukherjee, "WDM–based local lightwave networks — Part II: Multihop systems," *IEEE Network Magazine*, vol. 6, no. 4, pp. 20–32, July 1992.

[57] B. Mukherjee, S. Ramamurthy, D. Banerjee and A. Mukherjee "Some Principles for Designing a Wide–Area Optical Network." *IEEE INFOCOM'94*, Vol. 1, pages 1d1.1—1d1.10, 1994.

[58] S. Nugent, “The iPSC/2 direct–connect communications technology.” In *Proceedings of the 3rd conference on Hypercube Concurrent Computers and Application*, Volume 1, Jan. 1988.

[59] R. W. Numrich, P.L. Springer and J.C. Peterson, “Measuerment of Communication Rates on the CRAY-T3d Interprocessor Network”. In *Proceedings of High Performance Computing and Networking*, LNCS 797.

[60] R. Manchek, “Design and Implementation of PVM version 3.0”, Technique report, University of Tennessee, Knoxville, 1994.

[61] C. Qiao and R. Melhem, “Reconfiguration with Time Division Multiplexed MIN's for Multiprocessor Communications.” *IEEE Trans. on Parallel and Distributed Systems*, Vol. 5, No. 4, April 1994.

[62] C. Qiao and R. Melhem. “Reducing Communication Latency with Path Multiplexing in Optically Interconnected Multiprocessor Systems.” In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 34-43, January 1995.

[63] C. Qiao and Y. Mei, “On the Multiplexing Degree Required to Embed Permutation in a Class of Networks with Direct Interconnects.” In *IEEE Symp. on High Performance Computer Architecture*, Feb. 1996.

[64] R. Ramaswami and K. Sivarajan, “Optimal Routing and Wavelength Assignment in All–Optical Networks.” *IEEE INFOCOM'94*, vol. 2, pages 970–979, June 1994.

[65] A. Rogers and K. Pingali “Process decomposition through locality of reference.” In *Proc. SIGPLAN'89 conference on Programming Language Design and Implementation*, pages 69-80, June 1989.

[66] C. Salisbury and R. Melhem “Modeling Communication Costs in Multiplexed Optical Switching Networks”, The *International Parallel Processing Symposium*, Geneva, 1997.

[67] K.Sivarajan and R. Ramaswami, “Multihop networks based on de bruiji graphs,” *Proceedings IEEE INFOCOM'91*, Bal Harbour, FL, pages 1001–1011, April 1991.

[68] K.M. Sivalingam and P.W. Dowd, “Latency hiding strategies of pre–allocation based media access protocols for WDM phontic networks,” in *Proc. 26th IEEE Simulation Symposium*, pages 68 – 77, Mar. 1993.

[69] J. Stichnoth, D. O'Hallaron, and T. Gross “Generating communication for array statements: Design, implementation, and evaluation,” *Journal of Parallel and Distributed Computing*, vol. 21, no. 1, Apr, 1994, pp. 150-159.

[70] J. Subhlok, D. O'Hallaron, T. Gross, P. Dinda, J. Webb “Communication and memory requirements as the basis for mapping task and data parallel programs.” *Proc. Supercomputing '94*, Washington, DC, Nov. 1994, pp. 330-339.

[71] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross “Exploiting task and data parallelism on a multicomputer,” *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May, 1993, pp 13-22.

[72] S. Subramanian, M. Azizoglu and A. Somani, “Connectivity and Sparse Wavelength Conversion in Wavelength-Routing Networks.” *Proc. of INFOCOM’96*, pages 148–155, 1996.

[73] Stanford Compiler Group “The SUIF Library”, Stanford University.

[74] A. Sussman, G. Agrawal and J. Saltz, “PARTI primitives for unstructured and block structured problems.” *Computing Systems in Engineering*, Vol. 3, No. 4, pages 73–86, 1992.

[75] R.E. Tarjan “Testing flow graph reducibility.” *Journal of Computer and System Sciences*, 9:355-365, 1974.

[76] A.M. Vengsarkar “Optical Fiber Devices.” *Photonic Networks*, Springer, Pages 133–140, 1997.

[77] A. Venkateswaran and A. Sengupta “On a Scalable Topology for Lightwave Networks.” *IEEE INFOCOM’96*, Vol. 2, pages 4a.4.1—4a.4.8, 1996.

[78] X. Yuan, R. Gupta and R. Melhem, “Distributed Control in Optical WDM Networks,” *IEEE Conf. on Military Communications*(MILCOM), pages 100-104, McLean, VA, Oct. 21-24, 1996.

[79] X. Yuan, R. Gupta and R. Melhem, ”Demand-driven Data Flow Analysis for Communication Optimization,” *Workshop on Challenges in Compiling for Scalable Parallel Systems*, New Orleans, Louisiana, Oct. 23-26, 1996.

[80] X. Yuan, R. Melhem and R. Gupta “Compiled Communication for All–optical TDM Networks”, *Supercomputing’96*, Pittsburgh, PA, Nov. 1996.

[81] X. Yuan, R. Melhem and R. Gupta “Distributed Path Reservation Algorithms for Multiplexed All-optical Interconnection networks” *the Third International Symposium on High Performance Computer Architecture(HPCA 3)*, San Antonio, Texas, Feb.1-5, 1997

[82] X. Yuan, R. Gupta, and R. Melhem ” An Array Data Flow Analysis based Communication Optimizer,” *Tenth Annual Workshop on Languages and Compilers for Parallel Computing* (LCPC’97), Minneapolis, Minnesota, August 1997

[83] X. Yuan, R. Gupta, and R. Melhem ” Does Time Division Multiplexing Close the Gap Between Memory and Optical Communication Speeds?” *Workshop on Parallel Computing, Routing, and Communication* (PCRCW’97), Atlanta, Georgia, June 1997.

[84] X. Yuan, C. Salisbury, D. Balsara and R. Melhem, “A Load Balancing Package on Distributed Memory System and its Application the Particle-Particle Particle-Mesh (P3M) Methods.” *Parallel Computing*, Vol. 23, No.19, pages 1525-1544, Oct. 1997.

[85] X. Yuan, R. Melhem and R. Gupta “Performance of Multihop Communication Using Logical Topologies on Torus Networks.” *The Seventh International Conference on Computer Communications and Networks* (IC3N’98), Lafayette, Louisiana, 1998.

[86] X. Yuan and R. Melhem ”Optimal Routing and Channel Assignment for Hypercube Communication on Optical Mesh-like Processor Arrays.” *the Fifth International Conference on Massively Parallel Processing Using Optical Interconnections*(MPPOI’98), Las Vegas, June 1998

[87] X. Yuan, R. Melham, R. Gupta, Y, Mei and C. Qiao "Distributed Control Protocols for Wavelength Reservation and Their Performance Evaluation" Submitted to *IEEE trans. on Communications*, 1998.

[88] H. Zima, H. Bast and M. Gerndt. "SUPERB: A tool for semi–automatic MIMD/SIMD parallelization." *Parallel Computing*, 6:1-18, 1988.

[89] Z. Zhang and A. Acampora, "A Heuristic Wavelength Assignment Algorithm for Multihop WDM Networks with Wavelength Routing and Wavelength Reuse." *Proc. IEEE Infocom'94*, pp 534-543, June 1994.