COMPILER TECHNIQUES FOR EFFICIENT COMMUNICATIONS IN MULTIPROCESSOR SYSTEMS

by

Shuyi Shao

B.E. Computer Science and Engineering, Xi'an Jiaotong University, P.R.China, 1996

M.S. Computer Science, University of Pittsburgh, 2007

Submitted to the Graduate Faculty of

Arts of Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2010

UNIVERSITY OF PITTSBURGH

FACULTY OF ARTS AND SCIENCES

This dissertation was presented

by

Shuyi Shao

It was defended on

December 6, 2010

and approved by

Rami Melhem, PhD, Professor, Department of Computer Science

Alex K. Jones, PhD, Associate Professor, Department of Electrical and Computer Engineering

Sangyeun Cho, PhD, Associate Professor, Department of Computer Science

Youtao Zhang, PhD, Assistant Professor, Department of Computer Science

Dissertation Co-advisors: Rami Melhem, Professor, Department of Computer Science

Alex K. Jones, Associate Professor, Department of Electrical and Computer Engineering

COMPILER TECHNIQUES FOR EFFICIENT COMMUNICATIONS IN MULTIPROCESSOR SYSTEMS

Shuyi Shao, PhD

University of Pittsburgh, 2010

Technical advances have brought circuit switching back to the stage of interconnection network design for high performance computing. Although circuit switching has long connection establishment delays and the dedication of connections prevents other communicating nodes from sharing the network, it has simple control logic and significant cost advantage over packet or wormhole switching. With the proper assistance from compilers, circuit switching has the potential of providing significant performance benefits when connections can be established prior to the actual communication.

This dissertation presents a novel compilation framework for achieving efficient communications in circuit switching interconnection networks. The goal of the framework is to identify communication patterns in Single-Program-Multiple-Data (SPMD) parallel applications and compile these patterns as network configuration directives. This can significantly reduce the communication overhead on circuit switching interconnection networks.

A powerful representation scheme is developed in this research to capture the property of communication patterns and allow manipulation of these patterns. Based on the temporal and spatial localities of communications and the capability of the compiler to identify the communication patterns, we classify communication patterns into three categories – *static, persistent*, and *dynamic*. We target static and persistent communications, which are dominant in most parallel applications. To identify communication patterns, we develop a novel symbolic

expression analysis. We develop certain compiler techniques for analyzing communication patterns. Since the underlying network capacity is limited, we develop an algorithm to partition the program into phases based on the communication requirements and network capacity.

To demonstrate the effectiveness of our framework, we implement an experimental compiler. The compiler identifies the communication patterns from the source code, partitions the program into phases, and inserts the network configuration directives at phase boundaries to achieve efficient communications. The compiler also can generate communication traces, which provides useful information about the communication pattern correlated to the structure of the source code. We develop a multiprocessor system simulator to evaluate our techniques. Our simulation-based performance analysis demonstrates that using our compiler techniques can achieve the same level, or even better level of communication performance than fast packet switching networks while using much less expensive circuit switches.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

During the phase of this research I have been aided by a number of people. I take this opportunity to thank them for the various contributions that they have made to help me reach my goals.

I would like to express my deepest appreciation and thanks to my co-advisors, Dr. Rami Melhem and Dr. Alex K. Jones for their support, encouragement and guidance. Without them, this dissertation would never have happened. I am also grateful to the members of my thesis committee, Dr. Youtao Zhang and Dr. Sanyeun Paul Cho, for their review and suggestions concerning this research.

I would especially like to thank my families. Thanks to my parents for their love and overwhelmingly faith in my ability to achieve my goals. Thanks to wife, my lovely son and daughter for their unconditional love and being my motivation to finish.

Last, I would like to thank my fellow graduate students, the University of Pittsburgh and the cast of thousands that make the University of Pittsburgh such a wonderful place to learn. This research has been supported in part by NSF award number 0702452, the PERCS project at IBM, Defense Advanced Research Projects Agency (DARPA) under Contract NBCH3039004.

1.0 INTRODUCTION

This dissertation tackles the challenges of achieving efficient communications in circuit switched multiprocessor systems. It presents a novel compilation framework for identifying and exploiting communication patterns in parallel applications.

1.1 MOTIVATION AND BACKGROUND

High performance computing systems are being built out of ever-increasing numbers of processors [64, 2, 35, 22, 71, 66, 1]. For example, Roadrunner, the number one supercomputer on the top 500 list in 2009, has 129600 cores [1]. These large systems typically use packet or wormhole switching for inter-processor communication. However, as the system sizes increase, a scalable interconnect can consume a disproportionately high portion of the system cost in order to achieve low-latency and high-bandwidth communication. Although the quest for cheap, low-latency, high-bandwidth packet switching networks to interconnect large numbers of processors is worthwhile, circuit switching [17, 86, 11, 24] can be a cost effective alternative for achieving efficient communication in the high performance computing domain. By establishing direct connections between communicating processors, routing and buffering at intermediate switches can be eliminated, end-to-end protocols can be simplified, and both software and hardware overheads associated with data movement can be minimized.

However, circuit switching technology has two significant drawbacks. The first one is that the overhead of circuit establishment can be relatively large. The second one is that the number of connections that can be established simultaneously is limited. It comes from the fact that the resources used to establish a circuit are exclusively occupied by the circuit. Only when communications in parallel applications exhibit good localities, and these localities can be appropriately exploited, the benefits of circuit switching can outweigh its drawbacks. Using compiler techniques to optimize the communication in parallel applications – known as compiled communication [13, 84, 31] – becomes a promising approach for achieving efficient communications in the high performance computing domain. This approach is to infer the communication patterns of parallel applications at compile time and exposes them to an architecture, in which the multiprocessor systems' run-time components can rely on the compiler statically managing circuit switched interconnections.

The motivation of this work partially stems from the proposal to include an Optical Circuit Switching (OCS) network in the design of next generation high performance computing systems [8]. In the proposal long-lived bulk data transfers are routed through all optical circuit switching networks, which are characterized by high data rates but with high overhead for circuit establishment [25, 83, 8]. An OCS is less expensive than its electronic counterpart as it uses fewer optical transceivers. This interconnection technique is very effective if connections can be pre-established and the relatively long establishment overheads are amortized over the lifetime of the connections. Thus, it is crucial to know which connections will be used during each execution phase to achieve the expected communication performance and cost advantage on circuit switching enabled multiprocessor systems.

Note that, the communication-to-computation ratio of parallel applications can be very high. The communication time can consume more than half of the entire application execution time [107]. Qin et al. reported that communication-aware load balance can improve the performance of communication-intensive applications by up to 75.9 percent and 42.2 percent on average [108]. Generally speeding up communication performance can significantly improve the performance of parallel applications.

1.1.1 Switching Techniques in Multiprocessor Systems

The desire to enable circuit switching in multiprocessor systems is motivated by the investigation on the performance, availability, and cost of different switching techniques in current supercomputers. Circuit switching, packet switching and wormhole routing are three dominant switching methods that have been used in interconnection networks for high performance computing systems [21, 29, 39].

Circuit switching establishes a dedicated circuit between the source and the destination before the source processor can send data to the destination. Once a circuit has been established, the data movements between the source and the destination can happen at very high bandwidth and very low latency compared to packet switching techniques. This is mainly because circuit switching eliminates the needs of routing and buffering at all intermediate switches. Also, the end-to-end protocols can be simplified. However, the overhead of circuit establishment is relatively large. For example, in the design of OCS networks, the switches use optics in all elements of the data path. Setting up such circuits is typically accomplished through the use of MEMS-based (Micro-Electro-Mechanical Systems [25, 83, 8]) mirror arrays that physically move the light beam to establish an optical data path between any input and any output port. The

latency of circuit establishment is measured in milliseconds. Hence, only when the pattern of the communication is known in advance and the circuits can be effectively re-used, the overhead for establish connection can be amortized. Thus the expected benefits of including optical circuit switching in the OCS network can be realized.

In contrast to circuit switching, packet switching does not establish dedicated paths between the sources and destinations. Data, destination address, and other meta-data items are encapsulated into packets. The packets move along the switches and links until they arrive at their destinations or are dropped. The simplest packet switching technique is store-and-forward. Each switch independently determines where to send, and actually sends a packet out after it receives it. Each such step is called a hop and the sender may wait for an acknowledge message from the receiver. In this way, a packet incurs a delay which is at least proportional to the number of hops it has to travel since each switch along the path cannot start sending the packet out until the entire packet has arrived.

In order to reduce this dependence on distance (hops), wormhole switching [65] has been used. In this technique, a packet is further broken into small pieces called *flits* (*fl*ow control dig*its*). Instead of waiting until the whole packet is received, an intermediate switch can start sending a flit out whenever it is possible. In this way, a packet may spread over many switches on the path from the source to the destination. Wormhole routing was introduced in Ametec 2010 [29], a two-dimensional mesh. It has been widely used in a variety of multiprocessor systems including the Intel Paragon [10], Cray T3D [61, 43], iWarp [31, 28], IBM Power Parallel SP series [72, 5, 14, 81, 12], and the Quadrics switch [26]. Only when the first flit of a packet moves one step ahead or is consumed by the destination, the rest flits of the packet can move one step

forward. Virtual cut-through [45] is similar to wormhole routing except that the flits of a packet can pile up at an intermediate switch. This technique requires more queues at the switches.

As multiprocessor systems scale up, the cost of the interconnection network using packet or wormhole switching increases disproportionately and even dominates the system cost. Therefore, the cost advantage of circuit switching has drawn the interest of many researches in the high performance computing domain. Although not very popular, circuit switching has been used in some multiprocessor systems. For example, the Intel iPSC/2 and iPSC/860 use circuit-switched communication [6, 37, 78, 55]. NEC Earth Simulator [35] uses a statically configured circuit switching network, which has a huge electronic crossbar with 640x640 ports. Interconnection Cache Network (ICN) [33, 34] is similar to the Earth Simulator, in which each processing node has one dedicated channel. Circuit switching also exists in parallel systems which use passive optical components through time-division multiplexing (TDM [16, 30]), wavelength division multiplexing (WDM [23]) or a combination of the two [48, 49, 50].

Circuit switching hardware continues to improve due to improvements in technology. New technologies such as optical switching continue to be an alternative to electronic circuit switching. Optical switching provides several advantages such as capabilities to handle long wire lengths and achieve high bandwidths. However, the reconfiguration time of optical switching is relatively long compared to electronic switching (*ms* vs μs) [25, 83, 8]. Hence, techniques are needed to exploit the communication localities in parallel applications to amortize connection establishment overhead on circuit switching interconnect networks. When the communication in a parallel application can be predicted, ideally at compile-time, circuit switching becomes a promising alternative to packet/wormhole switching for efficient communication in multiprocessor systems.

1.1.2 Communication Locality and Analyzability

The feasibility of enabling circuit switching in multiprocessor systems relies on the existence of good communication locality in parallel applications. It has been observed that many parallel applications exhibit high degree of locality [41, 74, 8, 67, 69]. The interconnection networks can benefit from both temporal and spatial communication locality similar to memory systems exploiting locality of references through caches.

Temporal locality represents the effect of temporal aggregation of the inter-processor communications [63]. High temporal locality suggests that the execution of parallel application shows phases and communication aggregates in certain phases. The communication pattern in each phase may be different. But typically in each phase, inter-processor communication stably occurs across a set of connections, which is called a communication working set. This provides the opportunity to reduce communication latency by dynamically grouping and scheduling messages and pre-establishing a certain set of connections.

Spatial locality is determined by the distribution of the connections in the application and determines the size of the working set. It has been shown that each node tends to have only a small number of favored destinations for the messages it sends [47, 3, 4]. For example, the NAS parallel benchmark suite [7] exhibits very high spatial locality and therefore contains small working sets [3]. Although the maximal communication degrees of certain parallel applications can be very large, they can be effectively reduced for many applications to a small number with little performance loss by just filtering out a small percentage of messages [8]. Several research groups observed that the communications in many applications exhibit regular patterns [38, 52, 51, 8]. Additionally, it has been shown that these regular communication patterns can often be discovered through analysis of the source code [27, 13, 18].

Many previous efforts to analyze communications patterns in parallel applications are based on trace analysis [8, 10, 67]. However, traces can provide communication information only for a particular execution instance of an application on a particular platform. Using static compiler techniques can reveal the underlying communication pattern of a parallel application.

In summary, many parallel applications exhibit good communication locality and the locality can be effectively explored by compiler techniques to improve the performance of parallel applications. Thus we can enable circuit switching to take advantage of its cost benefits.

1.2 OVERVIEW OF THIS RESEARCH

The goal of this research is to develop compiler techniques that can achieve efficient communication in multiprocessor systems with circuit switching capabilities.

This research develops a compilation framework for analyzing the communication patterns in the parallel applications, in particular Message Passing Interface (MPI) [56] parallel applications. This compilation framework takes advantage of certain traditional compiler analysis techniques, such as control and data flow analysis, constant propagation, constant folding, and inter-procedural analysis, to collect information about MPI functions and program structure. Based on the collected information and using symbolic expression analysis, the framework identifies communications and partitions communication into phases. Finally, network configurations are generated for each communication phase and compiled into the application. These network configuration instructions are triggered at runtime to pre-establish circuits before actual communication appears.

This research classifies communication patterns into three categories: *static*, *persistent* and *dynamic*, based on the temporal and spatial locality of communications and the compiler capability to identify these localities. The classification implies different possibilities for reducing communication overhead in circuit switching networks. For example, given that the earliest opportunity for determining network configurations for a static communication operation is at compile-time, configurations may be statically inserted into the code by the compiler. This dissertation presents compiler techniques to identify static and persistent communication patterns. In particular, symbolic expression analysis is used to infer and compose symbolic expressions for the communication pattern. The need for symbolic expression analysis is highlighted by persistent communication because it contains variables that can only be represented by symbolic expressions and cannot be resolved at compile-time.

This research develops a powerful scheme to represent communication patterns. The representation describes collective and point-to-point communications using communication vectors and matrices, respectively. The vectors and matrices contain exact values if the communication pattern contains only static communications. Otherwise, they may contain symbolic expressions. One significant advantage of this scheme is that it has the capability to capture and represent the temporal locality by explicitly introducing communication phases in the pattern representation scheme. This scheme allows the manipulation of communication patterns through a set of convenient operations. It is also flexible and can be easily tailored to be used by other types of communication analysis.

Given that the communication working set of a parallel application may change during different phases of execution, an important aspect of the analysis of communication patterns is to identify and partition different communication phases. This research develops an algorithm for

partitioning communication phases, which uses the knowledge of program structure, symbolic expressions of logical connections, and network specifications. Based on the information, the algorithm segregates communication operations into communication phases with proper size for the parallel application. When generating network configuration from the communication pattern for a parallel application, this research uses an algorithm that can schedule as much communication as possible in the available networks when not all circuits can be implemented simultaneously. The application will be instrumented with network configuration instructions to pre-establish network connections during runtime.

This dissertation presents an experimental compiler that implements the compilation framework. This compiler is based on the SUIF compiler infrastructure [79] and integrates the compiler techniques developed in this research. Specifically, it can identify the communication pattern from the parallel application source code; compile the communication pattern as network configuration directives and instrument the application with these network configuration directives.

To evaluate the effectiveness of this research, it is essential to investigate the performance gains of this work in circuit switching enabled multiprocessor systems. Given the infeasibility of building a real system for cost reason and time limits in the context of this research, simulation approach is used. This research develops a multiprocessor system simulator. Our simulation-based experimental results demonstrate the usefulness of the compilation framework developed in this research.

In brief, this dissertation makes the following contributions:

• It extends the classification of communication as static, persistent, and dynamic, by separating persistent communication from traditional dynamic communication. This allows the compiler to tackle the majority of communications in typical parallel applications.

• It develops a compilation framework which integrates traditional and novel compiler techniques to exploit communication locality for achieving efficient communication in multiprocessor systems augmented with circuit switching capabilities.

• It presents a powerful and flexible communication pattern representation scheme which can effectively represent the temporal and spatial localities of communication.

• It develops novel symbolic expression analysis technique to analyze persistent communication patterns.

• It develops a communication phase partition algorithm to separate application into phases based on the underlying network architecture and communication requirement in the application.

• It develops an experimental compiler to implement the framework and a multiprocessor system simulator to evaluate the compiler.

• It explores and summarizes the generic and preferred runtime scheduling strategies for the proposed circuit switching enabled multiprocessor systems.

• It presents experimental study to show the usefulness and effectiveness of compiler techniques developed in this research.

1.3 ORGNIZATION OF THIS DISSERTION

The remainder of this dissertation is organized as follows. Chapter 2.0 presents prior work on studying communication characteristics. The relationship of this research and prior work is also discussed. Chapter 3.0 discusses the overall design of our compilation framework. It describes the machine and programming model that is targeted in the framework. It also describes the components and uses of the framework. Chapter 4.0 presents the classification of communication patterns and a powerful scheme to represent the communication patterns. It also shows the communications in NAS parallel benchmarks. Chapter 5.0 presents symbolic expression analysis for analyzing the communication patterns. Chapter 6.0 describes communication pattern manipulation and our phase partition algorithm. Chapter 7.0 discusses the runtime scheduling strategies. Chapter 8.0 describes the implementation details of our experimental compiler and the multiprocessor system simulator. Chapter 9.0 presents the experimental results to demonstrate the effectiveness of our compilation framework. Experimental results show that our compiler techniques can identify communication patterns. We simulate different types of multiprocessor systems. The performance comparison shows that the overall performance of parallel applications can be significantly improved by using techniques developed in this research. Conclusion, limitation and direction for future research are discussed in Chapter 10.0.

2.0 RELATED WORK

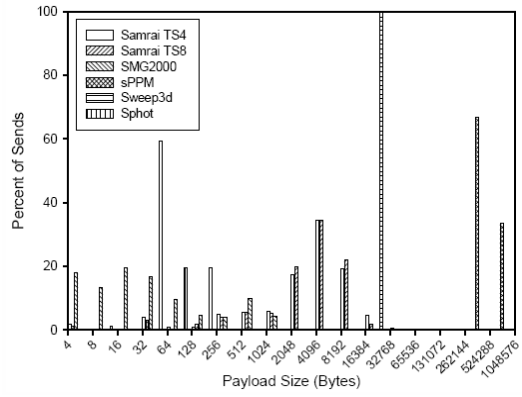
There is a wealth of prior related work that primarily comes from three areas of study. These areas are communication characteristics analysis, compiler techniques to exploit communication patterns, and simulation of multiprocessor systems. In the following sections, we discuss prior work and describe the relationship between them and this research.

2.1 COMMUNICATION CHARACTERISTICS ANALYSIS

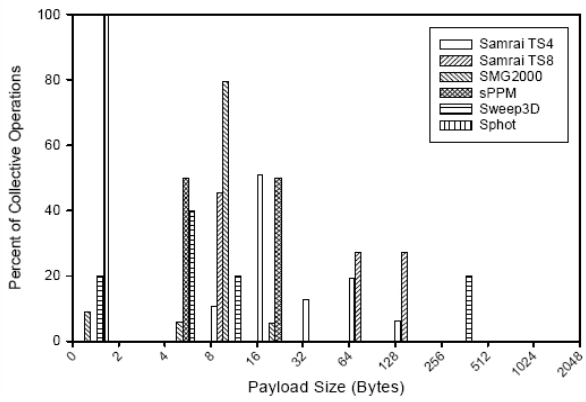
Understanding the communication pattern in the parallel application is critically important in the high performance computing domain. First, the characteristics of the targeted/expected workloads are important to determine multiprocessor system architecture. Second, system communication software needs to be optimized for both the system architecture and the application workload characteristics. Third, application developers need to make decisions on algorithms and data distribution models based on the system architecture and system communication software design. There is a lot of research on this topic.

In [77], Vetter and Mueller evaluated explicit communication characteristics across a set of diverse, large-scale scientific applications, primarily from the perspective of message passing via MPI. They use the MPI profiling layer to trace applications' communication activities. That is a trace-based approach. The MPI profiling layer is capable of capturing information about

each MPI call and the actual parameters of the function call. By focusing on the MPI activity of the applications, they presented the inherent communication signatures of a diverse range of applications. They observed that a trend of novel applications parting with regimented, static communication patterns in favor of dynamically evolving patterns. Note that they traced application communication activities with MPI's profiling layer. Another interesting observation from their work is that the payload size of point-to-point MPI messages tends to be large and the payload size of collective communication tends to be small. This suggests that enabling circuit switching in multiprocessor systems can benefit a significant portion of communication—point-to-point communication.



Payload size distribution for P2P messages (64 tasks).



Payload size distribution for collective communication(64 tasks).

Figure 1. Veter's MPI Payload Size Distribution

Faraj and Yuan [27] investigated the communication characteristics of MPI implementations of the NAS parallel benchmarks [7]. They used trace-based approach too. They instrumented MPI operations by implementing MPI wrappers that allow them to monitor the MPI communication activities at runtime. Then they examined the source code and marked each

of the MPI communication routines by hand. After the execution of the program, they analyze the trace files for all the nodes off-line to obtain the dynamic measurement of all communication activities. Their work also concluded that pure dynamic communication is only a small portion of all the communication in parallel applications. Thus a compiler has the potential of effectively optimizing communication statically for parallel applications.

Ho and Lin presented a work that statically analyzes communication structures in programs written in a channel-based message passing language called communication compiling component [38].

It is also worthwhile to mention the work of Ali et al. [104]. They presented performance models for certain collective communication algorithms that exploit features of the Cell architecture.

All the works described above use traditional trace-based approaches [87-91]. Trace-based approaches are limited to profiling and trace analysis. Although these attempts revealed many valuable characteristics of the communication in parallel applications, they do not provide a systematic way to identify accurate communication patterns with respect to connections. Trace-based approaches can only provide the communication information for particular execution instances of an application on a particular platform. When multiprocessor systems scales to massive number of processors, which can be thousands or even millions processors, trace-based approaches often become impractical. The overhead of trace generation may introduce unacceptable interference to the applications' behavior. The storage capacity limits may prevent researchers from collecting enough trace data. They often have to use sample-based analysis techniques. In short, there is lack of an automated compiler-based/compiler-assisted approach to reveal the communication patterns.

As CPU design goes into the chip multi processor (CMP) era, the computation within a single multi-socket machine naturally becomes parallelized. The data movements between different CMPs, caches, and memory modules may be very different. Compiler-based approaches can be used to improve these latencies too. For example, Jin and Cho utilized a software-oriented approach to optimize CMP cache management [103]. Although CMP research is out of the scope of this dissertation, they can share the same philosophy to improve system performance as what we used for MPI parallel application run.

2.2 COMPILER TECHNIQUES FOR OPTIMIZING COMMUNICATIONS

In the high performance computing domain, compiler techniques are developed to improve the execution of parallel applications, i.e. to shorten the executing time, optimize resource utilization, and reduce power consumption. We can also utilize compiler to schedule and optimize the communication activities of parallel applications if the compiler is given the knowledge of the interconnection network and is capable of identifying the application communication pattern at compile-time. This technique is called compiled communication [13, 27]. There are some interesting research efforts in this domain.

Liang et. al. presented a SUIF-based compiler and an adaptive System-On-a-Chip (aSOC) multi-core architecture [52]. Their compiler is actually an application translator that maps applications' high-level design representations to the cores on an aSOC chip. The compiler is based on SUIF infrastructure. It isolates code basic blocks from the SUIF intermediate representations. Basic blocks are the fundamental unit when they map an application to the cores. Assigning different basic blocks to different cores incurs different inter-core communication

requirements. They developed a heuristic cost-model to control the basic-block-to-core mapping. The mapping of each basic block is directed by the computation and communication heuristic. Once related basic blocks are assigned to specific cores, the inter-core communication needs are determined and communication instructions are generated and inserted by the compiler to control inter-core data transfer. Figure 2 described the architecture of Liang's system [52]. In the experiments, they mapped four high-bandwidth signal processing applications including an MPEG-2 video encoder and a Doppler radar processor to a prototype aSOC device using compiler assisted design mapping technology.

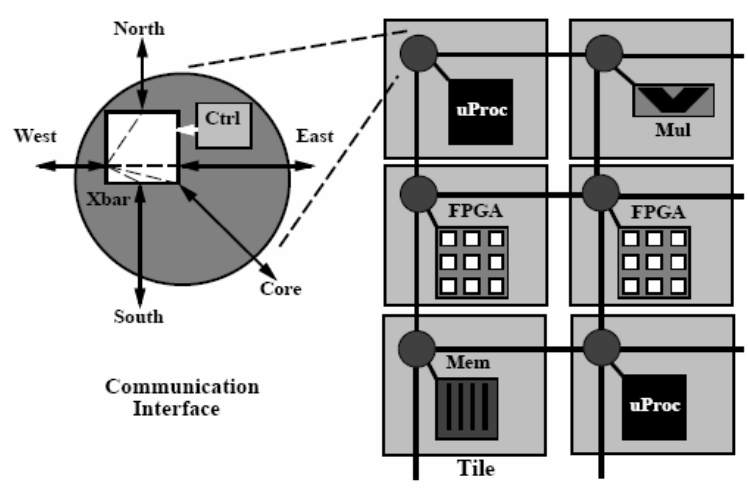


Figure 2. Liang's Adaptive System-on-a-Chip (aSOC) Architecture

Although the focus of the research of Liang et. al. is aSOC design, part of their application mapping techniques can be classified as compiled communication. But their work is in the scope of sequential program parallelization. Instead of identifying and compiling existing communication of parallel applications, they introduce inter-core communication by assigning different basic blocks of a sequential application to multiple heterogeneous resources, including cores.

Urfianto et. al. [105] presented another piece of work applying complier techniques on a Multiprocessor System-on-Chips (MPSoC) architecture. They utilized several optimizations provided by their compiler to implement efficient communication between processing elements. Their work is another example that demonstrates the feasibilities of applying compiled communication techniques on SoC architectures.

Shires et. al. [70] presented an algorithm for building a program flow graph representation of an MPI program. They augmented traditional program control flow graph with communication nodes and edges. They provided an interesting basis for important program analyses useful in software testing, debugging and code optimization. One limit of their approach is that their technique can only apply to static communication. It is possible, actually very often, that a communication occurs at run-time and has no associated communication edges. Thus it is infeasible to use their approach when the MPI program has a large number of control branches.

Cappello and Germain [13] exploited the association of compiled communication and a circuit switched interconnection network. They defined compiled communication as the technique to manage communication in parallel applications at compile-time. They investigated the feasibility of the compiled communication model and suggested using compiler to extract data references within parallel applications and translated the references to communication patterns. Their pioneer work pointed out a promising direction in interconnection network design and provided a fundamental theoretical performance analysis. But the proposed approach is targeted to data parallel programming language, which is a subset of (High Performance Fortran) HPF-like language [36, 75]. In that kind of programming languages, a communication pattern can be inferred from where the data are distributed and where the data are referenced.

Yuan [82, 85] studied compiled communication that can eliminate the run-time communications overheads of the dynamic communications by managing network resources at compile-time. A compiler, E-SUIF, is implemented to support compiled communications on optical TDM networks. In short, the techniques developed by the work analyze a program to determine its communication needs; then manage communication at compile-time with the knowledge of communication requirements and underlying network. The work targeted HPF-like programming models too. Those programs have programmer-specified explicit data alignments and distributions and no explicit communication. It is the duty of compilers to distribute the logical data set to different physical processors. Thus communication is needed to reference data stored on a remote processor.

The above two works did not target parallel programming models with explicit message passing. However, parallel applications with explicit message passing written in SPMD parallel programming models dominate today's high performance computing domain. These message-passing applications have potential to benefit from compiled communication techniques.

Karwande et. al. [44] made an attempt to apply compiled communication techniques in a prototype MPI library for Ethernet switched cluster. Basically they extends the MPI standard and uses separate routines for network control and data transmission and exposed network control routines to programmer for writing parallel programs which may benefit from compiled communication techniques. There are several limitations to the work. First, it requires the programmer, not just the compiler, to understand and manipulate network resources; and it needs run-time communication parameters to determine which implementation for a specific collective operation will be chosen at runtime. Second, the MPI system and applications may have to be modified from one cluster to another. Third, it handles collective communication only.

As we have described before, persistent communication is a significant portion of communication in parallel applications. Persistent communication can only be represented with certain unknown symbols which cannot be resolved at compile-time. This highlights the needs of performing symbolic analysis while analyzing and composing the communication pattern of an application.

Symbolic analysis is a useful technique in the parallel compiler research area. Several researches studied symbolic analysis techniques in the context of parallel compilation for High Performance Fortran (HPF)-like programs. Most of the work emphasizes the parallelization of programs with the assistance of symbolic analysis and a focus on loops and arrays. For example, Yuan et al. explored using compiled communication for HPF-like parallel applications as an alternative to dynamic network control [84].

In summary, to the best of our knowledge, there is no research that studies compiler techniques for efficient communication on circuit switching enabled multiprocessor systems for Single-Program-Multiple-Data (SPMD) programs, which is the subject of this dissertation.

2.3 SIMULATIONS OF MULTIPROCESSOR SYSTEMS

Simulation is a popular approach for research in the high performance computing domain. Many parallel simulations have been developed. High performance interconnection network simulations have different characteristics than general-purpose network simulators like NS [40], Parallel/Distributed NS [32], and GTSNetS [60]. To be able to simulate thousands of nodes, decisions regarding network topologies, routing algorithms, buffering, flow control, and so on,

have to be carefully made. Many simulators for parallel systems have been developed. Petrini and Vanneschi developed a simulator SMART and used it to simulate diverse traffic patterns for studying and analyzing different network architectures [59]. Benveniste and Heidelberger designed a conservative simulator of the IBM SP2 network [9]. MINSimulate [76] focuses on multistage interconnection, limiting it to a subset of well known network topologies. ´A la carte [42] is a Los Alamos computer architecture toolkit for simulating computing architectures. It is suitable to simulate extreme-scale systems (thousands of processors) with flexible architectural configurations.

An interconnection network simulator was presented by J Miguel-Alonso et.al. for MPI traces [106]. But their work was only able to handle traces generated by the MPI profiling mechanism PMPI. It cannot take any code artifacts, such as branches, loops, function boundaries, into consideration. Also they did not provide a way to emulate computation time. These facts make it an interesting related work to this research but we cannot take any advantage from it.

To summarize, none of the currently available simulators for parallel systems meet the needs for this research. First, our target simulated parallel systems must be able to include both circuit switching and packet switching interconnection networks. Second, a run-time component is needed to explore different communication scheduling strategies. Thus we need to develop our own multiprocessor system simulator.

3.0 OVERALL DESIGN OF THE COMPILATION FRAMEWORK

As described in Chapter 1.0, when parallel applications exhibit good communication locality and the locality can be effectively utilized, circuit switching has the potential of providing efficient communication. In this chapter, we present the overall design of our compilation framework to exploit the communication locality in parallel applications. First, we describe the machine model and programming model that are targeted in our framework. Then we describe the components in our compilation framework, including communication detection component, communication analysis component, communication compiling component and trace generation component. Finally, we describe the uses of our compilation framework.

3.1 MACHINE AND PROGRAMMING MODEL

To be able to exploit communication locality, we need to extract the communication patterns from application programs. The method to extract communication patterns in programs depends on both the machine model and the programming model.

There are some requirements for multiprocessor systems to be able to effectively utilize the communication locality in parallel applications. First, the interconnection network must have the capability to preload and keep network configurations. Because of the large connection setup delay in circuit switching, it is crucial for the system to establish circuits prior to the actual

communication requests. When the network cannot support all the needed circuits, dynamically setting up and tearing down circuits may lead to circuit thrashing, which establishes the same circuits many times during the execution of the application. Thus, keeping (or pinning) circuits in the network is often a better decision than dynamically setting up and tearing down circuits.

Second, we need to deploy a packet/wormhole switching network in the system in addition to the circuit switching network. The performance of this network can be significantly loosened to keep its cost low. The necessity of this network originates from the fact that there exists dynamic communication in some parallel applications. The time interval from when the needed circuits can be known to when the corresponding communication operation appears mostly is much shorter than the circuit establishment delay. For such cases, immediately delivering the message through even a slow packet/wormhole switching network may result in lower message latency than transferring the message through the circuits switching network.

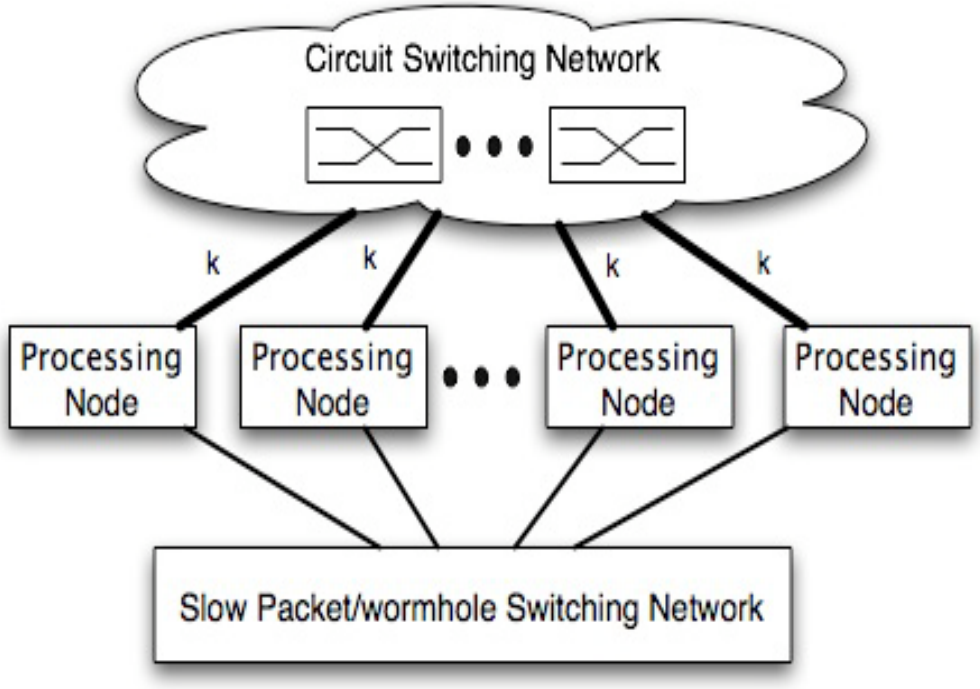


Figure 3. System Model of Targeted Multiprocessor Systems

Based on these requirements, Figure 3 gives an abstract model of multiprocessor systems that are targeted in our compilation framework. The system includes two kinds of interconnection networks: a circuit switching network and a packet/wormhole switching network. The circuit switching network can support *k* simultaneous connections to each processing node. It also allows connection preloading and reconfiguration.

In our framework, we target Single-Program-Multiple-Data (SPMD) programs with explicit message passing. In such programs, programmers explicitly use communication primitives to perform the required communication. The communication primitives can be high level library routines, such as Message Passing Interface (MPI), or low level communication primitives such as the shared memory operations in the CRAY T3D [92] and CRAY T3E [93]. Nowadays, SPMD is the dominant parallel programming approach.

3.2 COMPONENTS OF THE COMPILATION FRAMEWORK

The goal of our compilation framework is to identify communication patterns in Single-Program-Multiple-Data (SPMD) parallel applications and compile these patterns as network configuration directives. This can significantly reduce the communication overhead on circuit switching interconnection networks.

Figure 4 gives an overview of our compilation framework. Our framework includes four components: *communication detection component* to collect communication operations in the MPI parallel applications; *communication analysis component* to identify communication patterns and partition the communication phases; *communication compiling component* to insert

network configuration directives and *trace generation component* to automate trace generation for MPI applications. In the following sections, we describe these components.

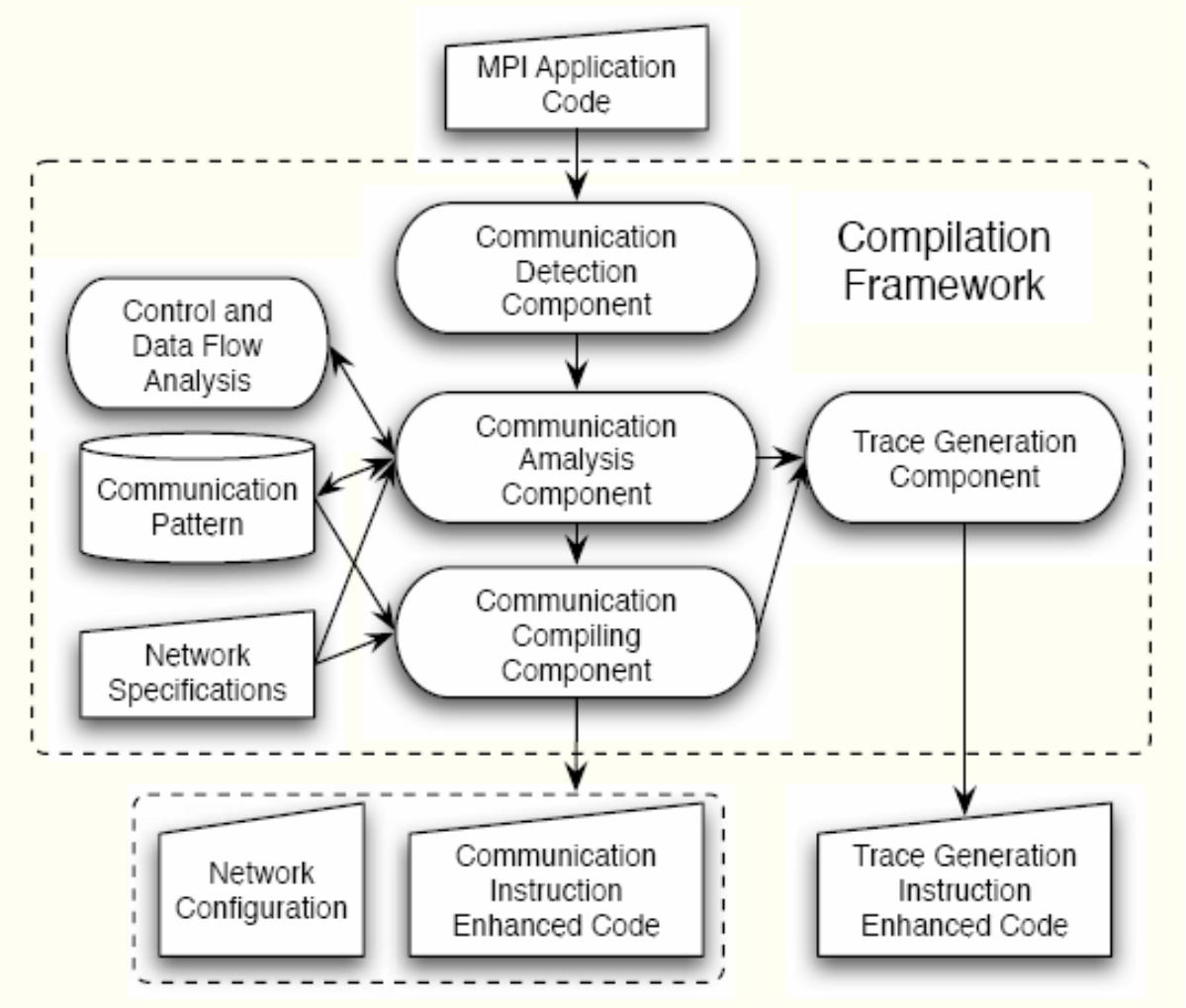


Figure 4. Overview of the Compilation Framework

3.2.1 Communication Detection Component

The communication detection component is responsible for identifying all explicit message passing functions (i.e., the MPI functions) used in an application and the related variables and parameters used in these MPI function calls. This is the very first step for further analysis. It provides information about which constants, which variables, and what part of code are involved in communication operations. We will use the sample code shown in Figure 5 to briefly demonstrate what the communication detection component does.

```
1 call mpi_init(ierr)
2 call mpi_comm_rank(MPI_COMM_WORLD, me, ierr)
3 call mpi_comm_size(MPI_COMM_WORLD, nprocs, ierr)
4 if (me .eq. 1) then
5   call mpi_send(buf, 16, MPI_INT, 0, tag, MPI_COMM_WORLD, ierr)
6 endif
```

Figure 5. Sample MPI Code Segment Where Node 1 Sends a Message to Node 0

The communication detection component will identify that there are 4 MPI functions in the code segment, `mpi_init, mpi_comm_rank, mpi_comm_size and mpi_send`. Line 1 is the MPI function which initializes the run-time MPI context for this node. One task of the initialization works is to coordinate with other nodes and allocate an integer-type rank for this node. The rank can uniquely identify a node within an MPI application execution instance.

Line 2 dictates the rank of the node is in variable *me* and Line 3 dictates that the total number of nodes running this application is in variable *nprocs*. Line 5 dictates that there is a point-to-point communication from node *1* to node *0* and the message size is 16 integers.

3.2.2 Communication Analysis Component

After the MPI operations and parameter information are identified, the communication analysis component identifies and analyzes the communication patterns based on a powerful representation scheme. Current compiler techniques, such as control and data flow graph (CDFG) analysis, are capable of inferring information to analyze the communication behavior of an entire program. Besides, we need advanced compiler techniques to analyze the communication patterns. For example, we need inter-procedural constant propagation to determine if the communications are static. Many communication operations utilize array parameters. For instance MPI operation *scatter* allows one node to distribute messages to many other nodes and *alltoallv* allows many nodes to send messages to many nodes. Such MPI operations use arrays to specify what nodes are involved in the operation and the data (type and count) related to each node involved. We need array section analysis to analyze these array operations. Also, we need strong symbolic expression analysis because persistent communications include variables that can only be represented as symbolic expressions.

Communication analysis component operates as follows. First, the CDFG is constructed for each procedure and communication related information is propagated through the graphs. Then a communication graph is built for the entire application. In this whole program communication graph, any CDFG node that does not affect the communication operations will be removed. The communication pattern, whose representation is detailed in Section 4.2, is

identified from the communication graph and stored in a communication pattern repository. The communication operations of the entire application are partitioned into communication phases. By mapping the communication patterns into a sequence of phases, it is possible to create more efficient communication working sets or groups of communications that occur in relatively close proximity. The granularity of the communication phases depends on the capacities of the communication network.

3.2.3 Communication Compiling Component

Based on communication patterns identified by the communication analysis component and the interconnection network specification, the communication compiling component inserts network configuration instructions into the application. This is a network-dependant component.

The design of the network configuration instructions relies on the target interconnection network. For static communication patterns, the compiler generates a network configuration file that can be used by the loader and inserts respective network configuration setup instructions in the program. For persistent communication patterns, the compiler inserts symbolic network configuration instructions that aim at pre-establishing the needed connections at run-time prior to the actual communications.

For the example shown in Figure 5 (line 4-6), because communication happens between node *1* and node *0*, we can insert a network configuration directive (*0, 1, 0*). The first parameter specifies which circuit switching network will be used to establish the connection when there are multiple circuit switching networks. The second is the source node and the last is the destination node. Here, (*0, 1, 0*) means using circuit switch network *0* to connect node *1* and node *0*.

3.2.4 Trace Generation Component

Although trace study has the drawback of emulating execution based on a specific set of data and parameters, many efforts to study the communication patterns of parallel applications are based on traces. We have a trace generation component in our compilation framework to automate trace generation for MPI applications. This component is responsible for inserting trace generation instructions for MPI functions within the applications. It also includes additional instructions that detail communication-related artifacts and constructs of the source code. We use the traces to verify the analysis results obtained by the compiler. Also, the traces can be fed into parallel system simulators to investigate the effectiveness of our framework.

3.3 USES OF THE COMPILATION FRAMEWORK

As previously discussed, circuit switching has the potential of providing efficient communication when the communication locality in the application can be identified and utilized. Our compilation framework can be used to identify and exploit the communication locality for parallel applications to achieve efficient communication.

Although we focus on MPI applications, our framework can be applied to other explicit message passing programming models written in SPMD style, where each processor independently executes the same program on its private data. To be able to work for other programming model, the part that needs to be changed in our framework is the detection of communication operations (i.e., instead of detecting MPI functions, our communication detection component needs to collect communication operations in other format).

The compiler techniques developed in our framework can benefit other research work in the high performance computing domain. For example, The Interconnection Cache Network (ICN)) [33, 34] statically configures its circuit switching crossbar according to the mapping from the communication graph in the parallel applications. Thus, the efficiency of obtaining the communication graph in the parallel application matters but it is beyond the scope of that work. ICN can benefit from using compiler techniques developed in our framework to obtain the communication graph in the parallel application efficiently.

We developed an experimental compiler to implement this compilation framework. The proposed components are implemented within the experimental compiler. We take advantage of Mathematica for symbolic expression manipulation. This significantly speeded up the research. This compiler is used on the benchmarks selected for this research.

Note that trace-based approach is still used by many researchers to study MPI applications. Our experimental compiler can automatically enhance MPIs programs with trace generation instructions for collecting traces of MPI operations and related code artifacts, e.g. function boundaries and branches. Thus it is a convenient tool for trace-based researches.

A multiprocessor system simulator is developed for this research. It takes execution traces and network configurations as input and generates accurate execution time information for performance study. The details will be provided at Chapter 8.2. This simulator can be used for studying MPI program and multiprocessor system.

4.0 COMMUNICATION PATTERN

Based on the communication locality and the compiler capability to identify the locality, we classify communication patterns into *static*, *persistent* and *dynamic*. This classification implies different possibilities to utilize the communication locality. In this Chapter, we first describe our classification. Then, we present a communication pattern representation approach to describe collective and point-to-point communications. This representation captures the property of communication patterns and allows manipulation of these patterns. Finally, we present the communications in NAS parallel benchmarks, which experimentally show that the majority of the communications are static and persistent.

4.1 COMMUNICATION PATTERN CLASSIFICATION

The communication within most parallel applications tends to exhibit good spatial locality and good temporal locality. The spatial locality dictates that each node often communicates only with a small number of other nodes. The temporal locality describes that the active communication set tends to be stable for certain time interval and then change to another set. Each such time interval is called a communication phase in this dissertation. Based on the temporal and spatial locality of communications and the compiler analyzability of the application code, we classify communications within a phase into three categories: *static*, *persistent* and *dynamic*. In this

context, a communication operation is specified by the source and destination of the messages exchanged. We refer to this as the topology of the communication.

Static - Communication is static if it can be completely determined through compile-time analysis. That is the compiler can identify both the temporal locality and the exact topology of the communication.

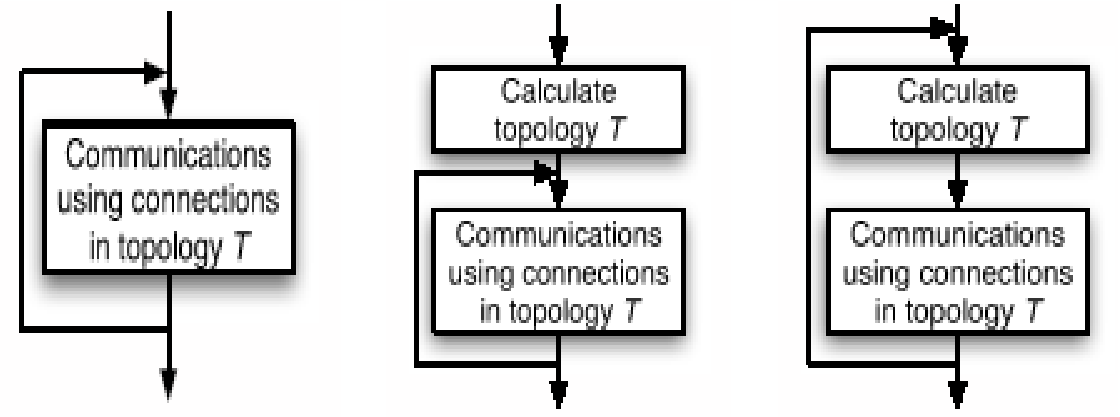
Persistent – Communication is persistent if, though the compiler cannot determine the exact topology of communication, it can determine that the topology does not change during the phase. That is, its temporal locality can be identified by the compiler, but its spatial properties remains unknown until run-time.

Dynamic – Communication is dynamic if it is neither static nor persistent.

Given that the communication working set of a parallel application may change during different phases of execution, an important aspect of the analysis of communication patterns is to identify and segregate different communication phases. It has been observed that a main source of communication temporal locality originates from loop structures of MPI programs. Hence, it is natural to consider a loop, which contains communications, as the basic building blocks of phases.

In Figure 6, we illustrate the definition of static, persistent, and dynamic communications when a phase is defined as a loop. Specifically, the communication operations are static if the topology (in terms of sources and destinations) can be completely resolved at compile-time, as in Figure 6 (a). In Figure 6 (b), the topology cannot be determined until run-time. However, once defined, the topology is repeatedly used within the loop. In this case, we call the communication

operations persistent. In Figure 6 (c), the communications are dynamic because during each iteration of the loop, the topology is re-calculated prior to use.



(a) Static: T is resolvable at compile-time.

(b) Persistent: T is calculated before a phase.

(c) Dynamic: T changes inside a phase.

Figure 6. Static, Persistent and Dynamic Communications

The above classification implies different possibilities for reducing communication overhead in circuit switching networks. For example, given that the earliest opportunity for determining network configurations for a static communication operation is at compile-time, configurations can be statically inserted into the code by the compiler at the phase boundaries. For persistent communication, the topology of the communication is not known at compile-time. However, it is possible to insert at compile-time symbolic expressions specifying the topology that may be resolved at run-time. By placing these symbolic expressions at the earliest point where the expression will be resolved, the network reconfiguration may still be able to take place prior to the use within a phase.

This classification enables us to target each class with the most appropriate network technology and operating mechanism. Specifically, static and persistent communications will be compiled and dispatched to the circuit switching interconnect network. Dynamic communications will be dispatched to the packet/wormhole switching network.

4.2 COMMUNICATION PATTERN REPRESENTATION

Previous researches have observed that the communication operations in many applications exhibit regular patterns [38, 53, 52, 8]. Additionally, it has been shown that these regular communication patterns can often be discovered through analysis of the source code [27, 13, 18].

To solve a particular scientific problem, a parallel application is often organized in computational phases. Given that these parallel applications have computational phases, we can expect that their communication behaviors are in a similar way. For example, the communication topologies of adaptive applications evolve during their execution time. Even for parallel applications that have static communication patterns, their active communication working set may change as the phases change. The result is one or multiple communication phases. Communication phases are not identical to computational phases, but are strongly associated with them. For example, some computational phases contain no communication and thus can be ignored when identifying communication patterns. Several computational phases may yield a single communication phase. The number of phases is an artifact of the analysis used to partition the communications into phases. To be able to perform communication analysis effectively, it is

necessary to have a flexible representation that can capture the properties of communication patterns accurately.

A traditional communication representation models the rough logical topology of communication patterns, (e.g., binary tree, 2-D mesh, hypercube). These representations are too coarse and cannot describe the communication topologies accurately. Another disadvantage of prior representations is that they cannot describe temporal information. Our representation scheme is designed specifically to avoid these limitations and to effectively represent the temporal and spatial properties of communication patterns.

In the following, we describe our representation scheme, which uses a communication matrix and vector pairs to describe the communication pattern in an application. The fact that the communication pattern of an application contains phases is important and must be described while representing the pattern.

We define all the communication operations of an application as a communication pattern. There are two types of communication operations in MPI applications: collective communications and point-to-point communications. In order to represent the collective communications, we define a c-enumeration to describe the set of collective communication functions invoked in a parallel application. Note that the participating nodes of a MPI program run can forms different communication groups—called MPI communicator, which will be detailed in Section 5.5. A node can participate multiple communicators. Each node within a specific communicator has a unique integer id—called *rank*. The same node may have different ranks within different communicators. The default communicator, referred to as MPI_COMM_WORLD retains all the processors.

Definition: A ***c-enumeration*** is a list of all the collective communications that appear in a parallel application. Each collective communication is represented by a pair, the function name and optionally the corresponding MPI communicator.

For the function names, we use AA, AV, AR, and RD to represent MPI_Alltoall, MPI_Alltoallv, MPI_Allreduce, and MPI_Reduce respectively. The communicator is omitted if it is the default MPI communicator. For each related MPI communicator, the same collective MPI functions have exactly one instance in the c-enumeration. Each communication pattern retains a unique c-enumeration.

Example 1: CE = {AA, AR, (AR, $commu_1$), RD} indicates that there are three different types of collective communications in the application. The first two and the last operations are performed in the default MPI communicator. The third operation, MPI_Allreduce, is performed in a user-defined communicator $commu_1$. The communication detection component of the framework is responsible for building *c-enumerations.*

Figure 7 formally describes the grammar to represent the communication pattern with one or multiple phases for parallel programs. In rule 2, col_i represents any collective MPI function, and $comm_i$ represents the corresponding MPI communicator. Here, we assume that m is the number of elements in *c-enumeration.*

Rule 1: *communication pattern* → *c−enumeration*, *phases*

Rule 2: *c−enumeration* → {(*col*$_0$, *comm*$_0$), ..., (*col*$_{m-1}$, *comm*$_{m-1}$)}

Rule 3: *phases* → ε | *phase phases*

Rule 4: *phase* → <*c−vector, p−matrix*> I [*phases*]

Rule 5: *c−vector* → ε | <w_0,w_1, ...,w_{m-1}>

Rule 6: *p−matrix* → ε | <*deterministic p−matrix*>| <*symbolic p−matrix*>

Figure 7. The Grammar for Communication Pattern Representations

As described in rule 3, a communication pattern consists of a sequence of phases. A phase may also be a loop of a sequence of phases that repeat in any execution instance of an application, represented by square brackets in rule 4 of the grammar. Rule 4 also indicates that a basic communication phase is described by a *c-vector* and a *p-matrix* that represent all the collective and point-to-point communications, respectively, in that phase.

Definition A ***c-vector*** corresponds to a *c-enumeration*. Each element of the vector represents the weight of the corresponding collective communication in the *c-enumeration*.

Definition A ***p-matrix*** is a N × N matrix that describes a set of point-to-point communications. The entry in position *i*, *j* describes the weight of communication from processor *i* to *j*. Certain weights may be unknown, which can be represented by **δ,** defined next.

Definition ***δ*** represents any unknown values, variable, vector, or matrix.

In the above definitions of *p-matrices* and *c-vectors* we do not enforce a specific meaning for the communication weight. Three options are described here: (1) a single bit value to indicate if there exists point-to-point communication from the source processor to the destination processor (2) the message volume or (3) message count. In the cases that the compiler cannot construct even a symbolic expression for a point-to-point communication, $\boldsymbol{\delta}$ is used in the symbolic expression for that matrix entry.

A *p-matrix* is **deterministic** if the total number of processors N is known and each entry of the *p-matrix* is a constant. A deterministic *p-matrix* is used to represent static communications. When the size N of a *p-matrix* is a symbolic constant and/or any entry can only be described by a symbolic expression instead of a constant, it is a **symbolic** *p-matrix*. A symbolic *p-matrix* can always be described by a formula list. Persistent communications can usually be described by symbolic *p-matrices*.

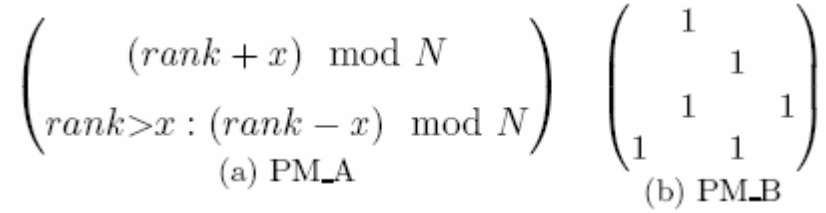


Figure 8. A Symbolic p-matrix and Its Deterministic Instance Where *x*=1 and *N*=4

Example 2: As shown in Figure 8, *PM_A* and *PM_B* are a formula list and a deterministic p-matrix, respectively. *PM_A* describes a communication pattern in which each processor *rank* sends to *rank+x* and *rank-x* if *rank-x* > 0 where *x* is determined at run-time and *N* is the total number of processors. In the case $x = 1$ and $N = 4$, a deterministic p-matrix *PM_B* is inferred from *PM_A*.

Table 1. The Communication Pattern of IS

c-enumeration	*{AR, AA, AV, RD}*	
Phases	*c-vector*	*p-matrix*
Phase 0	<1,1,1,0>	NULL
Phase 1	<1,1,1,0>	NULL
Phase 2	<0,0,0,1>	PM_IS

Example 3: The communication pattern of IS (integer sorting) program from NAS parallel benchmark suite [7] is shown in Table 1 and its deterministic *p-matrix* in phase 2 is shown in Figure 9.

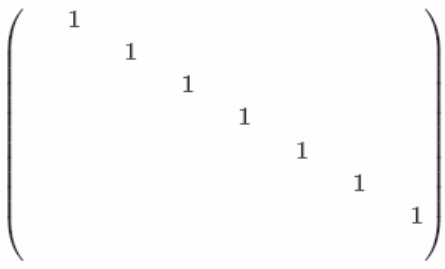


Figure 9. p-matrix for IS with 8 Processors

As we already described earlier in this section, programmers can create any user-defined communicator when necessary. A node may have different rank numbers in different communicators. And each point-to-point communication function specifies the source node or destination node in terms of rank and the corresponding communicator. Two send operations to two nodes with different ranks in two different communicators may use the same connection and should be represented in the same entry of a *p-matrix*. This is possible because all user-created

communicators are built from the default communicator with standard communicating group and communicator manipulating functions.

The communication pattern representation scheme can be used to represent both compile-time identified communication patterns and the communication patterns identified from execution traces. The main advantage of our pattern representation scheme is that it captures the time evolving, or phase, property of communication patterns.

4.3 COMMUNICAITONS IN NAS BENCHMARKS

We study the communications in the NAS Parallel Benchmarks v2.4.1 [7]. The NAS Parallel Benchmarks are a set of parallel programs designed to help evaluate the performance of parallel supercomputers. The benchmarks, which are derived from computational fluid dynamic (CFD) application, consist of five kernels and three pseudo-applications. They are widely used in researches in the high performance computing domain.

The distributions of static, persistent, and dynamic communications in these benchmarks are shown in the following tables. Table 2 shows the point-to-point communications and Table 3 shows the collective communication operations.

For the point-to-point operations, IS, CG and LU contain only static communications. MG, BT, SP contain only persistent communications. For BT and SP, the destination set for each node is calculated prior to all point-to-point communications and are used through application completion. For MG, there are two communication stages. In each stage, the destination set for

each node is calculated prior to the communications and is retained until each stage completes. These two stages contain multiple outmost loops.

Table 2. The point-to-point Communications in NAS Benchmarks

Benchmark	Static	Persistent	Dynamic
IS	100%	0%	0%
CG	100%	0%	0%
MG	0%	100%	0%
BT	0%	100%	0%
SP	0%	100%	0%
LU	100%	0%	0%

Table 3. The Collective Communications in NAS Benchmarks

Benchmark	Static	Persistent	Dynamic
IS	0.4%	0%	99.6%
CG	100%	0%	0%
MG	100%	0%	0%
EP	100%	0%	0%
FT	0%	100%	0%
BT	100%	0%	0%
SP	100%	0%	0%
LU	100%	0%	0%

The data in Table 3 were acquired through compile-time analysis with the exception of IS and FT for which the percentage of dynamic communications were obtained from an execution trace obtained on 128 processors. The reason is that IS and FT consist of more than one class of collective operations and this leads to different results with different execution configurations—problem size and number of processors. For FT, while the number of total nodes increases, persistent all-to-all communications dominate the message volume and the percentage of static communications is extremely low.

These data show that the majority of communications in parallel applications are static or persistent communications. We can use compiler techniques to identify these patterns and achieve efficient communications on circuit switching networks.

5.0 SYMBOLIC EXPRESION ANALYSIS

As described in Chapter 1.0, to effectively leverage circuit switching, compiler techniques are needed to identify the communication patterns in parallel applications. Traditional compiler optimizations manipulate constants or reorder code for faster execution. In this research, the goal of using compiler techniques is to analyze the communication patterns within a parallel application. In many cases, the expression for a communication destination or volume will contain variables that are not compile-time constant. Thus, we need compiler techniques that manipulate symbolic expressions to determine how these values are calculated.

```
void p(int k)
{
    int i;
    for(i=0; i<1000; i++) {
        if (myrank < nprocs/2) {
          MPI_Send(buf,1, MPI_INT_TYPE,
            (myrank+k)%nprocs,1000,COMM);

          MPI_Send(buf2, 1, MPI_INT_TYPE,
            (myrank+2*k)%nprocs,1000,COMM);
        } else {
          MPI_Send(buf,1,MPI_INT_TYPE,
            (myrank-k)%nprocs,1000,COMM);

          MPI_Send(buf2, 1, MPI_INT_TYPE,
            (myrank-2*k)%nprocs,1000,COMM);
        }
    }
}
```

Figure 10. Code Example for the Demonstration of Symbolic Expression Analysis

In this Chapter, we present the techniques of symbolic expression analysis to identify communication patterns. An example of a piece of MPI code is shown in Figure 10. The code includes a 1000-iteration loop which forms one communication phase. In each iteration, a node sends two messages. All the nodes are partitioned into two sets according to their communication behaviors—the lower half set and the higher half set. Each node in the lower half set sends one message to node`(myrank+k)%nprocs` and one message to `(myrank+2*k)%nprocs`. Each node in the higher half set sends one message to node`(myrank-k)%nprocs` and one message to `(myrank-2*k)%nprocs`, respectively. Note that the MPI standard requires that send operations and receive operations are paired. The sender has to always explicitly specify the receiver, while the receiver can take a message without specifying a particular sender. Thus we focus on send operations while analyzing communication patterns. So in Figure 10 we omit receive operations to simplify our explanation.

We use this example to show how our techniques work. Our analysis is based on control and data flow graph. Thus, we begin our discussion with the construction of the CDFG.

5.1 CONTROL AND DATA FLOW GRAPH

Control and data flow graph (CDFG) is used for many compiler analysis. A control flow graph (CFG) is a directed graph representation of all possible traversal paths within a program during its execution. In the graph each node represents a basic block. Each directed edge represents a possible control path (e.g. a branch or jump instruction). There are two special “pseudo-nodes” in a CFG–the entry block, which is the unique entry point to the entire flow graph, and the exit block, which is the unique exit point for leaving the flow graph.

The CFG for our example is shown in Figure 11. There are 10 basic blocks: BBEntry, BB2, ..., BB9, and BBExit. The back edge from BB5 to BB3 shows that there is a loop in the code. BB2 corresponds to the instructions initializing the loop index variable. BB3 is the basic block telling if the loop is finished or not. If the loop is not finished, the control goes to the loop body which starts from BB4; otherwise, it exits to BB6, then BBExit. The loop body is a branch structure. BB7 and BB8 are the then-part and the else-part, respectively. Each of them contains two MPI_Send functions.

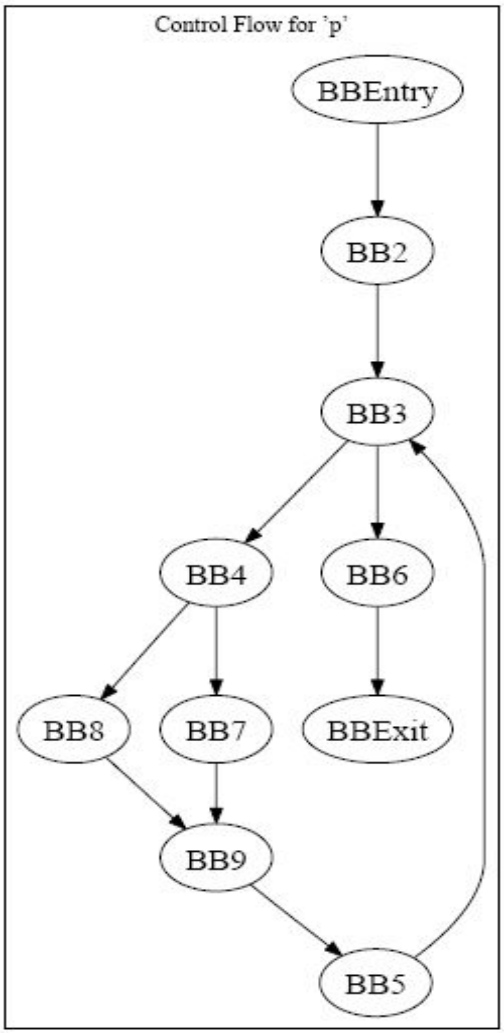


Figure 11. Control Flow Graph for the Example

Each basic block has a data flow graph (DFG). A DFG, which is a directed graph, represents the data dependencies within the code between control points. In a DFG, each node represents an operator (e.g. addition or logical shift) or an operand (e.g. a constant, a variable, or an array element). Each directed edge represents a data dependency that denotes the transfer of a value.

As we described before, BB7 is the *then-part* of the branch structure. Its DFG is shown in Figure 12. It shows how the parameters in MPI_SEND functions are computed.

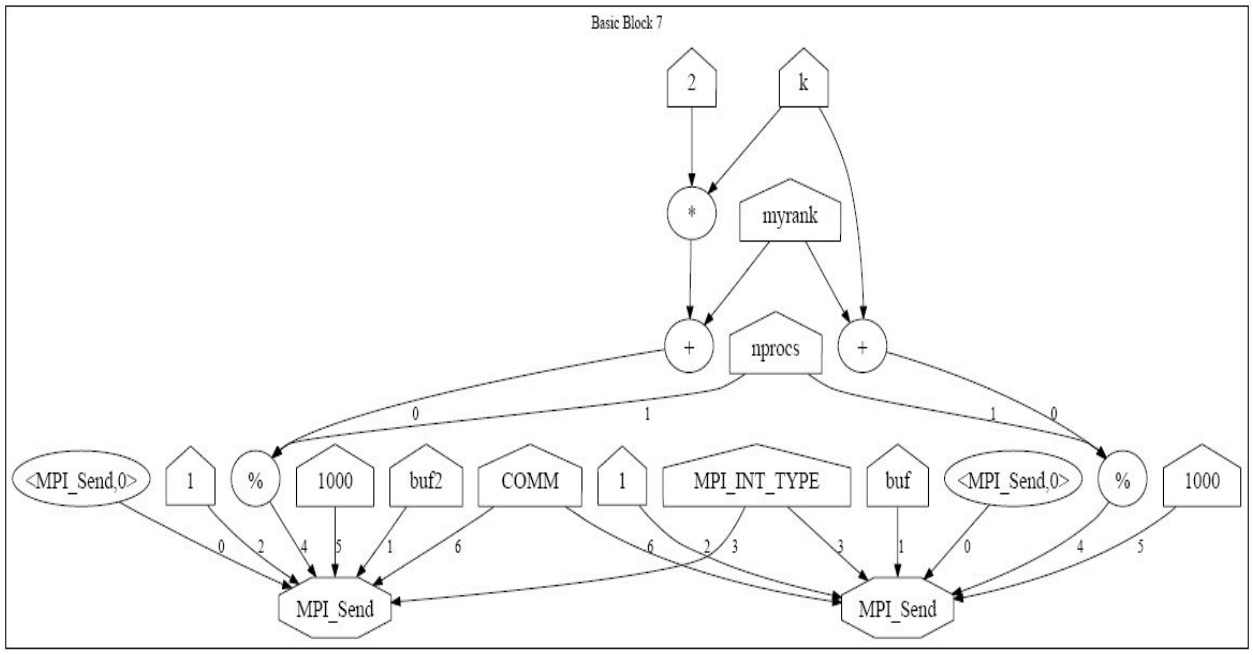


Figure 12. Data Flow Graph for BB7 in the Example

5.2 CONDITIONAL CONTROL FLOW

As MPI parallel programs are written in SPMD style, each process independently executes the same program on its private data. Often, different MPI processes take different execution paths depending on the evaluation results of the wrapping branch structures when the branch condition is based on the rank of the processes. In such situations, branch structures constrain the participating MPI processes taking different execution paths. In other words, a particular participating process only performs the communications in certain selected branches. Each connection in the communication pattern can be represented in the format: *source*, *destination*, *volume*, where *source* represents the source processes that participate in the pattern, *destination* is the destination expression, and *volume* is the expression for the size of the message. If a communication operation is not constrained by a condition related to the process id or rank, *source* is set to null to indicate all processes will perform the communication operation. Multiple conditions can be represented as $c_1 \wedge c_2 \wedge ... \wedge c_n$, each c_i represents a condition.

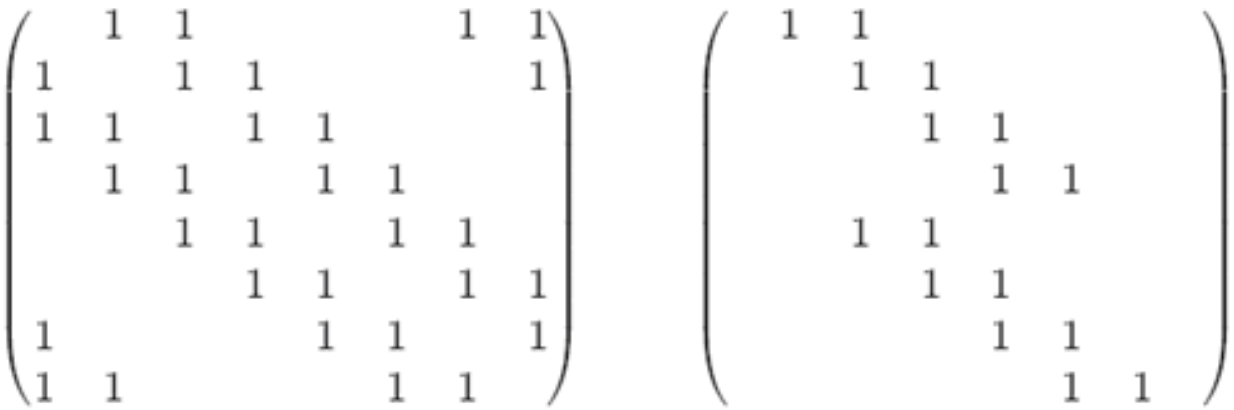
The example code from Figure 10 contains 4 MPI_SEND operations. Two of them are selected by the branch condition “myrank < nprocs/2”, while the others are constrained by “myrank >= nprocs/2”. Assume that we do not successfully take the branch condition into consideration or the branch condition is not related to the process rank, the *source* of every connection is set to *null*, which means all processes are involved in the connection. The symbolic representation of the communication matrix, A, is shown below. Note that each MPI_Send operation in Figure 10 sends 1000 integers (4000 bytes) in the message. So the *volume* of each connection is 4000.

$$A = \begin{pmatrix} \{null, (myrank + k)\%nprocs, 4000\}, \\ \{null, (myrank + 2*k)\%nprocs, 4000\}, \\ \{null, (myrank - k)\%nprocs, 4000\}, \\ \{null, (myrank - 2*k)\%nprocs, 4000\} \end{pmatrix}$$

When taking the branch condition into consideration (i.e., flow sensitive), the actual symbolic communication matrix, B, is:

$$B = \begin{pmatrix} \{myrank < nprocs/2, (myrank + k)\%nprocs, 4000\}, \\ \{myrank < nprocs/2, (myrank + 2*k)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank - k)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank - 2*k)\%nprocs, 4000\} \end{pmatrix}$$

At run-time the values of the variables in the symbolic expressions are resolved. For the values *nprocs*=8 and *k*=1, the corresponding communication matrices for symbolic matrices A and B are shown in Figure 13(a) and Figure 13(b), respectively. Note that because the source field in symbolic matrix A is null, it is necessary to calculate the destinations for each process. However for symbolic matrix B, we perform the same operation limited to those processes that satisfy the source condition. Because of taking control flows into consideration, we have more accurate communication patterns.



(a) Resolved matrix A. (b) Resolved matrix B.

Figure 13. Communication Matrix for the Example

5.3 TRAVERSAL ALGORITHM

The symbolic expressions analysis starts from each communication operation in the CDFG. The variables representing the destination and the volume are determined. Then we traverse the CDFG from the communication operation backward and build the symbolic expression. When branches are nested, the expressions representing the source are combined together during the traversal of the graph. The traversal completes when all the interested symbolic expressions have been resolved.

We assume that the MPI program is well structured and there are no goto instructions. Even with this restriction, loop structures still introduce cycles in the CFG, which create problems for traversing the CDFG. To determine the symbolic expressions it is necessary to remove cycles. To solve this problem, we take advantage of the information from the abstract syntax tree and treat all the CFG nodes in a loop as an individual super node. Thus, our CDFG becomes a directed acyclic graph. The processing of loop structures will be detailed in the next section.

As we have described, our target is to analyze the communication pattern in MPI parallel applications. We are only interested in the sources, destinations, and volumes of the exchanged messages. Because the CDFG of a MPI application is huge, analyzing everything can be very expensive. Thus, our symbolic expression analysis is driven by the target variables/expressions in communication operation functions. We only generate symbolic expressions for the target variables/expressions in terms of constants, MPI process rank, communicator size, and variables

which will be assigned value at run-time by input operations. Different MPI processes may take different execution path at run-time depending on the branch statements. We also need to perform symbolic expression analysis for the variables that appear in the conditions of the branch structures to be flow sensitive.

In brief, our symbolic expression analysis traverses the CDFG bottom up to compose symbolic expressions. During the traversal, we perform backward substitutions until the symbolic expressions of the targets becomes only in term of constants, MPI process rank, communicator size, and variables which will be assigned value at run-time by input operations.

As previously mentioned, each basic block in the CFG is represented by a DFG. Hence, we complete local symbolic expression analysis for each DFG prior to traversing the CDFG. Each DFG is by definition a directed acyclic graph. The values from input variables flow down through the graph through a number of operator nodes and finally converge at the output nodes.

We keep several lists to assist the symbolic expression analysis for each DFG:

(1) An input list *i* that contains all operands used for the calculation of output expressions in the current DFG;

(2) An output list *o* that contains all output nodes defined in the current basic block and the parameter variables/expressions used in communication operation functions;

(3) A function list *f* that contains all DFG nodes that represent a call instruction;

(4) A connection list *c* that contains references to all symbolic connections, each of which is in the format “source, destination, volume”, extracted from communication operation functions, which are detailed in Section 5.2; and parameter variables/expressions in the communication operation functions;

(5) A bypassing list *p* is used to holds bypassing information while traversing the CDFG.

An item in each list contains two parts: the variable name and the symbolic expressions. The symbolic expression field is calculated during the within-DFG symbolic expression analysis. The list i and list o are built during the construction of the CDFG; the list p and list c are initialized to empty.

The within-DFG symbolic expression analysis is completed through backward traversal along the DFG edges starting from each of the output nodes until it reaches the input nodes. Figure 14 presents the details of the algorithm.

Consider the example DFG from Figure 12. Our algorithm starts from MPI_Send call node. The destination operation for this node is the 4th parameter, which comes from the "%" node. Since "%" node is not a leaf node, we traverse upward to start building an expression from the "+" node and "nprocs" node. "nprocs" node is a leaf node. We traverse upward from the "+" node. After traversing to the "*" node, the final expression becomes "(2 * k + myrank) % nprocs", which only contains constants, MPI process rank and input variables.

Figure 15 describes the algorithm for global symbolic expression analysis. It traverses the CFG bottom-up from the program exit basic block using breadth-first search. The program exit block is represented as BBExit. Q is a queue of CFG nodes, each of which is either a basic block or a super-node representing a loop structure. C is a list of references of symbolic connections.

```
1 do_DFG(CFG_Node B)
2   mark all DFG nodes in B as un-visited
3   foreach DFG node n in the output list do
4     generate_symbolic_expr(n)
5
6 symbolic-expression generate_symbolic_expr(n)
7   if n is visited then
8     return the symbolic expression of n
9   if n represents a call instruction then
10    foreach predecessor p that is a parameter of n do
11      generate_symbolic_expr(p)
12    switch type of n:
13    case unary-operator:
14      p=the predecessor of n
15      s1=generate_symbolic_expr(p)
16      return operator.s1
17    case binary-operator:
18      p1=the 1st predecessor of n
19      p2=the 2nd predecessor of n
20      s1=generate_symbolic_expr(p1)
21      s2=generate_symbolic_expr(p2)
22      return s1.operator.s2
23    case variable or constant:
24      return variable name or value
25  mark n as visited
```

Figure 14. Pseudo Code for Within-DFG Symbolic Analysis

Consider the example from Figure 10. We begin from the BBExit node. As this is a pseudo-node, BB6 becomes our new node as it is the predecessor of the exit node. BB3-6,7-9 contains a loop, and as such is combined into a super block called BB3'. Thus symbolic expressions in BB6 are defined in terms of input variables into BB6, these variables are output variables from BB3' with symbolic expressions associated with them. As we traverse upwards, the symbolic expressions from BB3' and BB6 are combined. BB4,7-9 denote an if-then-else structure. During the creation of the expressions for BB3', these expressions are resolved as described in Figure 14. The expressions from BB9 are in terms of variables defined in either BB8 or BB9 or both. Thus, the expressions are replicated as described in line 20 based on the condition. The traversal continues until BB2 and BBEntry are reached.

We treat the loops as a super block to remove the cycle and allow the BFS to terminate when the entry node is reached. In the next section, we describe how the symbolic expression of an iterative structure is determined.

```
 1 mark all CFG nodes as un-visited
 2 add all predecessors of Bexit to Q
 3 mark Bexit as visited
 4 while (Q is not empty) do
 5   remove a node n from Q
 6   if (n is a basic block) then
 7     global_do_DFG(n)
 8   else
 9     process loop /*detailed in Section 5.4*/
10     foreach predecessor p of n do
11       if (all successors of p except n are visited) then
12         add p to Q
13         propagate partial symbolic expressions from n to p
14     mark n as visited
15
16 global_do_DFG(n)
17   add all items in list c to C
18   foreach symbolic expression e in n do
19     substitute unresolved variables in e with symbolic expressions for the outputs of n
20   if (n is branch test) then
21     augment true condition to the source field of connections from then part
22     augment false condition to the source field of connections from else part
```

Figure 15. Pseudo Code for CDFG Traversal Algorithm

5.4 GENERATING SYMBOLIC EXPRESSIONS FOR LOOPS

A large portion of MPI communication operations are embedded in loop structures. We see this characteristic in both benchmark applications such as the NAS parallel benchmarks, COMOPS, etc. as well as full blown parallel applications. This is not surprising as the loop structures in an application can often be classified as a phase of application in which communication configuration is persistent for a relatively long time. This justifies that the communication information from loop analysis is a good candidate to configure the network. Additionally, for our symbolic expression analysis, the destination variables of MPI communication operations are usually defined and calculated in loop structures.

The goal of our communication destination analysis is to acquire a list of symbolic expressions for destination variables in MPI calls for each loop iteration or/and a final symbolic expression for destination variable after the whole loop structure has finished executing. We have two kinds of loops in our CDFG—counted loops, whose iteration number is deterministic statically, and DO-WHILE loops, which terminates based on the condition following the loop body.

We target counted loops, i.e., FOR loops in C and DO loops in Fortran. The reason that we focus on counted loops is that we can determine their symbolic expressions through static loop analysis.

5.4.1 Static Loop Analysis

Note that loop structures form circles in the CDFG. Normal graph traversal algorithms, i.e. DFS or BFS, may fall into infinite loop on these circles. Hence we use an analytic approach when

processing loops while we perform symbolic expression analysis along the CDFG. We observe that the number of our symbolic analysis targeting variables, such as destination variables and volume variables, is typically very limited in each benchmark. We also find that these variables are usually only set up once in the program and normally will be rarely changed latter on, which together give us the chance to analyze these destination variables one by one.

As we have stated, a loop structure in the CDFG is treated as an individual super-node. Our loop analysis targets each such communication-related super-node which contains information about that loop including the upper bound, lower bound, loop index, step, and loop condition.

For each loop structure, we scan the loop body to see if it contains any definitions of communication destination variables. These variables fall into two classes: ones that are dependent on loop index and those that are not. We handle each of these situations differently.

If the variables are independent of loop index, our goal is to get a final symbolic expression after all loop iterations. In such case, we can leverage static symbolic loop solving functionality in Mathematica. Mathematica is a fully integrated environment for technical and scientific computing, especially strong in symbolic expression manipulation and calculation.

Mathematica provides an API interface called Mathlink which allows users to communicate with the Mathematica kernel in other programming languages. Our compiler feeds Mathematica via MathLink all the loop information and the non-iterative symbolic expression generated for destination variables. Mathematica's do loop command will automatically sort the loop information, iterate over the loop iterations and return a symbolic expression for input variables after loop calculation. As an example, if we have the following code segment:

```
for (int i = 1; i <= 3; i++)
   y = y^2 + i
```

Our compiler feeds the following expression to Mathematica to represent the structure of the loop.

```
y = y0; Do[y = y^2 + i];{i, 1, 3, 1}; y
```

Note that the expression follows the requirements of Mathematica. The 1st clause indicates that y_0 is the initial value to y; The 2nd Do clause describes the computation within each iteration. The 3rd clause,{I, 1, 3, 1}, dictates that the loop index variable is i, whose lower bound value is 1, upper bound value is 3, and loop step is 1. The last clause specifies the target expression to compute, namely y.

The result returned from Mathematica is:

$$3 + (2 + (1 + y_0)^2)^2)^2$$

If the destination variables depend on the loop index, e.g. the destination is defined as an array dest[i], each element of the array may correspond to a different communication destination In this case, it is necessary to generate a symbolic expression for each loop iteration with the help of Mathematica. First, we generate symbolic expressions of the destination variables in the CDFG with the loop index unchanged. The initial value of the loop index and its symbolic expression as they increment between iterations are easy to obtain since we already recorded all the loop information. We feed all of these expressions to Mathematica through MathLink in loop order N times, here $N = \frac{loop\_upperbound - loop\_lowerbound}{loop\_step}$. As a result, Mathematica will return a list of symbolic expressions for the destination variables for each loop iteration.

5.5 COMMUNICATOR AND RANK

During the communication symbolic expression analysis, the operands representing communicators and the ranks of MPI processes need special attention. In a MPI programs, the communicator determines the scope of the “communication universe” in which communication take place. Each communicator contains a group of valid participants – MPI processes. The source and destination of a message is identified by process rank within the communicator. Hence, the symbolic expression analysis needs to know all the participants in the communicator. This can be achieved through keeping track of the group and communicator functions in the MPI program. In most cases, there is one default communicator appearing in the MPI program, MPI_COMM_WORLD, which contains all the MPI processes in the MPI program execution.

The rank of a MPI process is a unique ID of that process in a specific communicator and it can be obtained through calling the MPI_Comm_Rank() function. This function may be called many times in a MPI program and the value of the rank may be stored in different variables. During symbolic expression, we need to keep track of all these variables and treat them uniformly. Similar to the rank, in a MPI program, we need to know the size of communicator, which is used to determine the communication topology.

Note that both the rank and the size of a communicator are defined in the context of a particular communicator. All user-defined communicators can be constructed from the default communicator. By examining the group and communicator manipulation functions in a MPI program, it is possible to map a user-defined communicator to the default one. Then, the symbolic expressions of the sources and destinations are in the default communicator.

5.6 SUMMARY AND DISCUSSIONS

Note that, our symbolic expression analysis needs to cross the procedural boundaries. With the support of a crossing procedural boundary technique (i.e., on-demand inline), we bridge the gap introduced by the procedures and make procedural boundaries transparent to symbolic expression analysis. Thus when we do symbolic expression analysis, the CDFGs traversal has a logical view of no procedure boundary limits. Our symbolic expression analysis is different than traditional inter-procedural analysis since we are only interested in variables involved in the communication operations. For example, consider the procedure p(k) from Figure 10, which contains an instruction to send a message to *myrank+k* or *myrank-k*, and p is called from two different locations, L1 and L2, with parameters A and B, respectively. If A is known at L1, while B is not known at L2, then the communication within p is static when p is invoked from L1, while it is not if p is invoked from L2. This means that our analysis is context-sensitive. Operators in a code instruction will be treated differently when reached following each different path, as the destination variables in p can be analyzed to an explicit value when reached from L1 while it is unknown when reached from L2. However, traditional static analysis usually only conclude that the destination variables in p are constant when the variables can obtain the same constant value when reached from both L1 and L2.

Note that the symbolic expression analysis can not only compose symbolic expressions for sources, destinations, and volumes of messages, but also perform constant propagation and constant folding. The symbolic expression analysis is inter-procedural. Thus, the functionality of inter-procedural constant propagation and constant folding is already carried out.

6.0 COMMUNICATION PHASE PARTITION

Due to the limit of underlying network capacity, a multiprocessor system cannot support arbitrary communication patterns. When the network capacity is insufficient to establish all the circuits required by an application, it is necessary to divide the program execution into phases and schedule connections within each phase.

As described in Chapter 4.0, our communication pattern representation is able to capture the timely sequential property of communication phases, and allows the manipulation of communication phases. In this chapter, we first describe communication phase manipulations using our communication pattern representation. Then we present an algorithm to partition communication phases. At last, we present a case study to demonstrate how our phase partition algorithm works.

6.1 PHASE MANIPULATIONS

The execution of a parallel application can be divided into multiple phases based on communication behaviors. The active connections in a communication phase can be properly implemented in the underlying interconnection network. The interconnection network needs to be reconfigured at phase boundaries to establish all or as many as possible of the circuits needed by the corresponding phase.

Clearly, two conflicting criteria need to be considered for phase partition. On one hand, a communication phase should be small enough to allow the interconnection network to accommodate the communication requirements within the phase. On the other hand, a communication phase should be large enough to avoid the overhead of reconfiguring the network at unnecessary high frequency. Thus, the goal of the phase formation is to determine the largest communication working set that can fit within the capacity of the underlying circuit switching network and that requires the least reconfiguration during execution.

We take a bottom up approach to determine the optimal communication phases. First, we assume that each loop in the application is a phase. Then we manipulate these initial phase decisions to group the communications into new phases that are best suited to the capacity of the network. For example, we combine two adjacent phases if the network capacity is large enough to realize both of them; we remove some infrequent connections if the newly combined phase is beyond the capacity of the network. Communications over these removed connections will be delivered through the packet switching network. Using the communication pattern representation described in Chapter 4.0, we can easily manipulate the communication patterns in each phase.

We provide three core operations to manipulate communication phases.

Merge combines the *p−matrices* and *c−vectors* of two adjacent phases of a communication pattern into a new phase. If binary communication weights are used, this is equivalent to an OR operation. If traffic volume is used as weights, this is equivalent to a matrix/vector addition when we combine two *p-matrices*/*c−vectors*.

Filter removes connections below a threshold from a phase of a communication pattern.

Unwrap is the equivalent to unrolling a loop and merging all of the resulting phases into a single phase. This is particularly useful to deal with nested loops or adjacent loops that logically should be in the same phase.

We use an example to demonstrate the usefulness of these operations and the scenarios to use them. Assume there is an application whose communication pattern consists of 2 phases: the 1st phase is represented by *p-matrix* A; the 2nd phase is a 64-iteration loop, each iteration of which contains 2 phases, represented by B_1 and B_2 respectively.

$$A \begin{pmatrix} 0 & 327680 & 327680 & 327680 \\ 128 & 0 & 0 & 0 \\ 128 & 0 & 0 & 0 \\ 128 & 0 & 0 & 0 \end{pmatrix}; \; [\, B_1 \begin{pmatrix} 0 & 4096 & 0 & 0 \\ 16 & 0 & 4096 & 0 \\ 0 & 16 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, B_2 \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 16 & 0 \\ 0 & 4096 & 0 & 16 \\ 0 & 0 & 4096 & 0 \end{pmatrix}]$$

Given that B1 and B2 are directly adjacent phases, we can *Merge* them to a single *p-matrix* B_{12}.

$$\text{Merge}(B_1, B_2) \rightarrow B_{12} \begin{pmatrix} 0 & 4096 & 0 & 0 \\ 16 & 0 & 4112 & 0 \\ 0 & 4112 & 0 & 16 \\ 0 & 0 & 4096 & 0 \end{pmatrix}$$

Then we can *Unwrap* the loop and obtain a flattened *p-matrix* B for the second phase.

$$\text{Unwrap}(B_{12}) \rightarrow B \begin{pmatrix} 0 & 262144 & 0 & 0 \\ 1024 & 0 & 263168 & 0 \\ 0 & 263168 & 0 & 1024 \\ 0 & 0 & 262144 & 0 \end{pmatrix}$$

Now the communication pattern of this application is converted to

$$A \begin{pmatrix} 0 & 327680 & 327680 & 327680 \\ 128 & 0 & 0 & 0 \\ 128 & 0 & 0 & 0 \\ 128 & 0 & 0 & 0 \end{pmatrix} \text{ and } B \begin{pmatrix} 0 & 262144 & 0 & 0 \\ 1024 & 0 & 263168 & 0 \\ 0 & 263168 & 0 & 1024 \\ 0 & 0 & 262144 & 0 \end{pmatrix}.$$

To eliminate connections with low traffic, we apply *Filter* operation on the communication pattern. Assume the threshold is 10000, the resulting communication pattern becomes A' and B'. We can easily generate network configuration instructions to establish connections for such communication pattern.

$$A' \begin{pmatrix} 0 & 327680 & 327680 & 327680 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ and } B' \begin{pmatrix} 0 & 262144 & 0 & 0 \\ 0 & 0 & 263168 & 0 \\ 0 & 263168 & 0 & 0 \\ 0 & 0 & 262144 & 0 \end{pmatrix}.$$

Besides the above three core phase manipulation operations, there exist other potential types of phase manipulation operations. But not all of them are needed in the context of this research. For example, the *insertion* of communication phases is not needed because the initial phases we have are the loop structures from the original source. We never need to insert a phase to a communication pattern. Because our *c−vector* and *p−matrix* representations do not include temporal information, it is not possible to *split* one phase into two. But, we can always re-start the analysis from partitioning the original application to obtain finer-grain phases. If desired, a *delete* operation can be implemented through the use of the Filter and Merge operations. When applying these operations on the phases, the information about dynamic communications (represented by δ's) is ignored. This is a reasonable decision because compiled communication techniques cannot handle dynamic communication in general. All the operations are defined on a given communication pattern *CP*. In the following, we describe the syntax and semantics of the three core operations.

Definition: Merge($phase_i$, $phase_j$) :

Parameters: $phase_i$ = <$c-vector_i$, $p-matrix_i$>, $phase_j$ = <$c-vector_j$, $p-matrix_j$>. $phase_i$ and $phase_j$ are adjacent phases.

Semantics: $phase_i$ and $phase_j$ are merged into a new phase.

$phase_{new}$ = <$c-vector_{new}$, $p-matrix_{new}$> where

$c-vector_{new} = c-vector_i + c-vector_j$ and

$p-matrix_{new} = p-matrix_i + p-matrix_j$.

The + operation depends on the definition of the weights in the c−vector and p−matrix. For example, if the weights are binary bits in a deterministic p−matrix, the + operation is equivalent to an OR operation.

Definition: Filter(*phase*, *T*) :

Parameters: *phase* = <*c−vector*, *p−matrix*>, *T* is a threshold.

Semantics: This operation replaces *phase* in the communication pattern by a new phase in which any entry with value less than T in *c−vector* or *p−matrix* will be set to 0.

Definition: Unwrap(*phase*) :

Parameters: *phase* = [<*c−vector*, *p−matrix*>]. Assume the loop body of phase has been merged into a single phase <*c−vector*, *p−matrix*> as above.

Semantics: This operation unwraps the loop indicators of *phase*. If the communication weights are defined as single bit values, this operation uses one <*c−vector*, *p−matrix*> to replace a sequence of <*c−vector*, *p−matrix*>. If the weights are defined as message volume or message counts, the weights in the resulting phases

are multiplied by the loop iteration number compared to the weights in <*c−vector*, *p−matrix*>.

These three operations are the most import mechanisms to manipulate communication phases. Other supportive operations will be described when we present our communication partitioning algorithm. The algorithm uses these operations to determine the optimal phases so that the largest communication working set in the phase can fit within the capacity of the network and the least network reconfiguration during execution is needed.

Although we focus on handling point-to-point communication in this research, certain MPI library such as MPICH implements the collective communication operations based on point-to-point communication operations. Also, supercomputers, for example bluegene/L from IBM [2], often have a dedicated collective communication network which can efficiently perform collective communication. For such systems, collective communication can be entirely left to the collective networks. While clusters typically have only one network, in most cases an Ethernet network, connecting all the cluster nodes. Thus each collective communication has to be implemented as a number of point-to-point communication operations. For example, Karwande and Yuan studied how to effectively use point-to-point communication operations to implement collective communication operations [4].

The above argument suggests that sometimes the collective operations may appear as a set of point-to-point operations whose connections may overlap with some connections in the *p−matrix*. Thus an additional *C*ollective to *p*oint-to-point (*C2P*) operation is defined to decompose collective operations into a matrix of point-to-point operations as follows:

Definition: C2P(*phase*, *fmask*, *TR*) :

Parameters: *phase* = <*c−vector*, *p−matrix*>.

fmask is a bit-vector corresponding to *c-vector*. A "0" indicates that the corresponding collective communication in the *c-vector* will not be considered in the transformation. $fmask_i$ is the i^{th} bit of *fmask*.

TR = $<T_0, ..., T_{m-1}>$ where each T_i in TR is a matrix which converts the i^{th} collective communication operation to one or more point-to-point communications represented by a *p−matrix*.

Semantics: This operation transforms the *fmask* selected collective communications in *phase* to point-to-point communications represented by a number of p−matrices and adds these p−matrices to the original p-matrix to generate $p\text{−}matrix_{new} = fmask_0 \times w_0 \times T_0 + ... + fmask_{m-1} \times w_{m-1} \times T_{m-1}$. The remaining collective communications in *c−vector* form $c\text{−}vector_{new}$.

Let us demonstrate an example of using C2P operation to translate a broadcast operation to a number of point-to-point communications. Assume a communication pattern has *c-enumeration* <REDUCE, BCAST, GATHER>. We use binary weights within this example. A specific phase *P* of the communication is given below:

$$c\text{-}vector = <0, 1, 0>;\ \ p\text{-}matrix = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$

Assume we implement the BCAST in a binary tree structure, the communication operation sequence is *SEQ* = {0→0, 0→2; 0→0, 0→1; 2→2, 2→3}. The *fmask* is set to 010.

Here we only need to give T_2 of the translation matrix vector TR. T_2 is $\begin{pmatrix} 1 & 1 & 1 & \\ & & & 1 & 1 \\ & & & \end{pmatrix}$, which includes and only includes connections required by *SEQ*. After applying the C2P operation, phase *P* becomes $\begin{pmatrix} 1 & 1 & 1 & \\ & & & 1 & 1 \\ & & & \end{pmatrix} + \begin{pmatrix} & 1 & & \\ & & 1 & \\ & & & 1 \\ 1 & & & \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 1 & 1 & \\ & & 1 & \\ & & 1 & 1 \\ 1 & & & \end{pmatrix}$. The resulting *p-matrix* satisfies all the communication requests needed by phase *P*.

6.2 PHASE PARTITION ALGORITHM

The communication phases within an application typically become known at run-time. However, the structure of the source code can often provide enough directions to determine reasonable phase lengths and boundaries. Loops play a key role in the determination of phases in parallel applications [19] and work similarly for communication phases. Thus, we use loops and the code blocks between loops as the starting point to build phases. Adjacent phases whose joined communication pattern do not violate the capability of the network can be merged together to form larger phases.

The control structures arising from branch structures create difficulties in merging adjacent phases. We break down branches into two categories, rank-dependent and data dependent. Rank-dependent branches are the most difficult structures to handle as some processors execute each branch, concurrently. Thus, our solution in this case is to merge all phases contained into a single phase and to filter out lowest-bandwidth connections until the

resulting pattern can fit into the network. However, in some cases it may be possible to determine optimized patterns for each branch depending on static knowledge of the condition.

In data-dependent branches, all processors will take either one direction or the other. This condition holds because only branches containing communication operations are considered. If data dependent conditions allow different processors to follow different branches containing communication operations, it would be problematic to have matching sends and receives. Thus, for these branches, communication patterns can be merged within branches, but need not necessarily be merged across branches as is necessary with rank-dependent conditionals.

In addition to the core operations from Section 6.1, we also need several other basic phase operations. They are described below.

Definition: Adjacent(P_j, P_k) returns true if P_j and P_k are two directly adjacent phases that are not separated by loops or conditional boundaries and returns false otherwise.

Definition: **FilterUntilFit**(*P*, *NET*) removes certain connections from the communication pattern of phase *P* until *P* fits in network *NET*. The algorithm to implement this operation is detailed in the next Section 6.2.1.

Definition: CanBeMerged(P_j , P_j, *T*, *NET*) returns true if the two phases, P_j and P_k can be merged and fit into the network *NET* without violating the user specified parameter *T*. *T* is a threshold on the maximum weight of a connection that can be filtered.

Our phase partition algorithm is presented in Figure 16. The code starting with line 1 constructs the initial phases of the application by merging the adjacent basic-phases that fit into the network. Starting in line 7, all phases in the rank-dependent branch structures are merged and the communication pattern is filtered until it fits into the network, regardless of the values of T.

Starting in line 13 and continuing through the end of the pseudo-code, the algorithm revisits merging phases within loops. First, loops that contain a single phase are unwrapped at line 14. Starting at line 15, when possible, data-dependent branch structures are flattened into single phases. Finally, at line 18 adjacent phases are re-examined in case newly unrolled loops or merged branch structures have created phases that can be merged.

The above algorithm iterates until the phases reach a steady state creating the final phase partition of the application. Communication instructions can now be placed into the code prior to the execution of each phase.

```
 1 Create a basic phase for each MPI communication function call
 2 do
 3   foreach phase P do
 4     foreach phase Q such that Adjacent(P, Q)==1 do
 5       if CanBeMerged(P, Q, T, NET) then Merge(P, Q)
 6 while(phases can be merged)
 7 foreach rank-dependent branch structure BS do
 8   foreach branch Bi of branch structure BS do
 9     foreach phase P in Bi do
10       foreach phase Q in Bi such that Adjacent(P, Q)==1 then Merge(P, Q) into Pi
11   for any two branches Bi, Bj in BS containing phases Pi, Pj,
         respectively, Merge(Pi, Pj) into PBS
12   FilterUntilFit(PBS, NET)
13 do
14   foreach loop L if L contains a single phase PL then Unwrap(PL)
15   foreach data-dependent branch structure BS do
16     for any two branches Bi, Bj in BS such that each contains a single
         phase, Pi, Pj , respectively do
17       if CanBeMerged(Pi, Pj, T, NET) then Merge(Pi, Pj)
18   foreach phase P do
19     foreach phase Q such that Adjacent(P, Q)==1 do
20       if CanBeMerged(P, Q, T, NET) then Merge(P, Q)
21 while (phases can be merged)
```

Figure 16. Phase Partition Algorithm

6.2.1 FilterUntilFit Operation

In Section 6.2, we have presented the operation primitive FilterUntilFit(*P*, *NET*), which removes connections from the communication pattern of the phase *P* until *P* fits in network *NET*. It reduces the density of a communication matrix to fit it in the network and thus impacts the performance of our partition algorithm. One of the most straight-forward algorithms for FilterUntilFit is to remove the lowest weight connection one by one until the entire communication pattern can fit into the network. This is a greedy algorithm which does not necessarily produce optimal results.

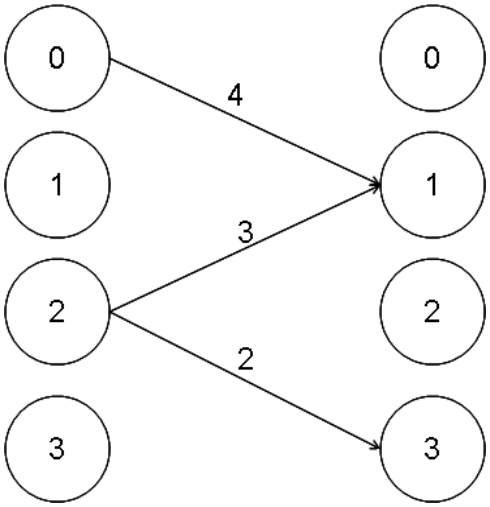


Figure 17. A Communication Pattern *P*

We model a communication matrix as a bipartite graph. That is, in the graph we have two disjoint sets of vertices: one set of vertices represent the source nodes, the other set of vertices represent the destination nodes. When a processor does both send and receive, it is represented by a vertex in the source set and a vertex in the destination set, respectively. An edge which connects one source vertex and one destination vertex represents the traffic originating from the

corresponding source node to the corresponding destination node. There is no edge connecting two vertices both of which belong to the source set or the destination set.

For example, assume that our network can implement 2 permutations simultaneously. Assume that there is a communication pattern *P* as shown in Figure 17, where number of each edge represents the communication between a source-destination pair. Using the greedy algorithm, connection <2→3> will be removed first, then <2→1> gets removed. When we apply FilterUntilFit operation on pattern *P*, the resulting *P'* will contains only one connection <0→1> and its weight is 44.4% of the original Pattern *P*. But the optimal solution is to separate <2→1> from the other two connections. The resulting pattern *P''* contains all required connections which are organized in two permutations (<0→1>, <2→3>) and (<2→1>).

According to König's theorem [94], if the maximum vertex degree of a bipartite graph is N, a minimal edge coloring uses N colors. If we assign the communication edges with the same color to the same circuit switch, we can satisfy that communication pattern with N circuit switches. Thus, FilterUntilFit operation becomes to select *n* colors from N colored bipartite graph, where *n* is the network capacity, N is the maximum communication degree of the pattern, and *n* < N. There are many matured algorithms doing bipartite graph edge coloring [95, 96, 97, 98]. We use a slightly revised implementation which is inferred from the original proof of König's theorem and detailed in Figure 18.

```
 1 do
 2   pick the highest weighted uncolored edge e(src, dst)
 3   if (exist color c not used by both src and dst) then
 4      color e with c
 5   else
 6       pick color c2 used by src not by dst, the edge is e2(src, dst_c2)
 7       pick color c1 not used by src
 8       find longest path p, p includes e2, each edge on p has color c2 or
             c1 and no adjacent edges have the same color
 9       change color of each edge on p by c1->c2 and c2-> c1
10 while (exist uncolored edge)
```

Figure 18. Edge Coloring Algorithm

Note that, in the above algorithm, we always pick the edge with the highest weight to color. This decision makes the heavy traffic connections aggregated together. Thus we can fit a communication pattern into a specific network by filtering out less percent of communication.

6.3 CASE STUDY

In this section, we present a case study to demonstrate how to use our algorithm and phase manipulation operations to obtain communication patterns. Our example is shown in Figure 19, which extends the sample code shown in Figure 10. In this section, we focus on demonstrating our phase partition algorithm. In the first part of the case study, we assume network capacity is 3.

```
 1 void p(int k)
 2 {
 3   int i;
 4   for(i=0; i<1000; i++) {
 5       if (myrank < nprocs/2) {
 6           MPI_Send(buf,1, MPI_INT_TYPE,
 7                    (myrank+k)%nprocs,1000,COMM);
 8           MPI_Send(buf2, 1, MPI_INT_TYPE,
 9                    (myrank+2*k)%nprocs,1000,COMM);
10       } else {
11           MPI_Send(buf,1,MPI_INT_TYPE,
12                     myrank-k)%nprocs,1000,COMM);
13           MPI_Send(buf2, 1, MPI_INT_TYPE,
14                    (myrank-2*k)%nprocs,1000,COMM);
15       }
16    }
17 }
18 main()
19 {
20      ...
21      /* stage 1 */
22      p(5);
23      ...
24      /* stage 2 */
25      p(4);
26      ...
27      /* stage 3 */
28      p(2);
29      ...
30      /* compu global value x and broadcast to all nodes */
31      if (x > NO_WORK) {
32          p(1);
33      }
30      ...
31  }
```

Figure 19. Code Example for Phase Partition Case Study

BEGIN

Step 1: Run line 1 of the algorithm in Figure 16. Each individual MPI communication call is identified as one basic communication phase. We can identify 16 basic phases from the source code.

```
 1   for(i=0; i<1000; i++) {
 2       if (myrank < nprocs/2) {
 3           ({null,(myrank + 5)%nprocs,4000})
 4           ({null,(myrank +10)%nprocs,4000})
 5       } else {
 6           ({null,(myrank – 5)%nprocs,4000})
 7           ({null,(myrank –10)%nprocs,4000})
 8       }
 9   }
10
11   for(i=0; i<1000; i++) {
12       if (myrank < nprocs/2) {
13           ({null,(myrank + 4)%nprocs,4000})
14           ({null,(myrank + 8)%nprocs,4000})
15       } else {
16           ({null,(myrank – 4)%nprocs,4000})
17           ({null,(myrank – 8)%nprocs,4000})
18       }
19   }
20
21   for(i=0; i<1000; i++) {
22       if (myrank < nprocs/2) {
23           ({null,(myrank + 2)%nprocs,4000})
24           ({null,(myrank + 4)%nprocs,4000})
25       } else {
26           ({null,(myrank – 2)%nprocs,4000})
27           ({null,(myrank – 4)%nprocs,4000})
28       }
29   }
30   if (x > NO_WORK) {
31     for(i=0; i<1000; i++) {
32       if (myrank < nprocs/2) {
33           ({null,(myrank +1)%nprocs,4000})
34           ({null,(myrank + 2)%nprocs,4000})
35       } else {
36           ({null,(myrank –1)%nprocs,4000})
37           ({null,(myrank – 2)%nprocs,4000})
38       }
39     }
40   }
```

Step 2. Run line 1~6. Among the 16 basic phases, Adjacent() return TRUE only for the pair of phases in each branch. For each adjacent pair of phases, CanBeMerged() returns TRUE, this is because that each phase has a communication degree[1] 1 and the network capacity is 3 which can satisfy 3 connections. Thus we merge each pair of adjacent phases and generate 8 bigger phases.

[1] The communication degree of a pattern is the maximal value of number of nodes each node communicates to.

```
 1   for(i=0; i<1000; i++) {
 2       if (myrank < nprocs/2) {
```
$$\begin{pmatrix}\{null,(myrank+5)\%nprocs,4000\},\\ \{null,(myrank+10)\%nprocs,4000\}\end{pmatrix}$$
```
5       } else {
```
$$\begin{pmatrix}\{null,(myrank-5)\%nprocs,4000\},\\ \{null,(myrank-10)\%nprocs,4000\}\end{pmatrix}$$
```
 8       }
 9   }
11   for(i=0; i<1000; i++) {
12       if (myrank < nprocs/2) {
```
$$\begin{pmatrix}\{null,(myrank+4)\%nprocs,4000\},\\ \{null,(myrank+8)\%nprocs,4000\}\end{pmatrix}$$
```
15       } else {
```
$$\begin{pmatrix}\{null,(myrank-4)\%nprocs,4000\},\\ \{null,(myrank-8)\%nprocs,4000\}\end{pmatrix}$$
```
18       }
19   }
21   for(i=0; i<1000; i++) {
22       if (myrank < nprocs/2) {
```
$$\begin{pmatrix}\{null,(myrank+2)\%nprocs,4000\},\\ \{null,(myrank+4)\%nprocs,4000\}\end{pmatrix}$$
```
25       } else {
```
$$\begin{pmatrix}\{null,(myrank-2)\%nprocs,4000\},\\ \{null,(myrank-4)\%nprocs,4000\}\end{pmatrix}$$
```
28       }
29  }
30  if (x > NO_WORK) {
31    for(i=0; i<1000; i++) {
32       if (myrank < nprocs/2) {
```
$$\begin{pmatrix}\{null,(myrank+1)\%nprocs,4000\},\\ \{null,(myrank+2)\%nprocs,4000\}\end{pmatrix}$$
```
35       } else {
```
$$\begin{pmatrix}\{null,(myrank-1)\%nprocs,4000\},\\ \{null,(myrank-2)\%nprocs,4000\}\end{pmatrix}$$
```
38       }
39    }
40  }
```

Step 3. Run line 7~12. Branches at code line 2, 12, 22, 32, and 42 are rank-dependent branches, we will perform merge operations to obtain one individual phase for each of them. Each resulting phase has a communication degree of 2, which is less than the network capacity. Line 8~11 of the algorithm produces results as below. The FilterUntilFit() at algorithm line 12 makes no change. The branch at line 30 is a data-dependent branch, and remains untouched in this step.

```
 1   for(i=0; i<1000; i++) {
```

$$\begin{pmatrix} \{myrank < nprocs/2, (myrank+5)\%nprocs, 4000\}, \\ \{myrank < nprocs/2, (myrank+10)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank-5)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank-10)\%nprocs, 4000\} \end{pmatrix}$$

```
9   }
11   for(i=0; i<1000; i++) {
```

$$\begin{pmatrix} \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000\}, \\ \{myrank < nprocs/2, (myrank+8)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank-4)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank-8)\%nprocs, 4000\} \end{pmatrix}$$

```
19   }
21   for(i=0; i<1000; i++) {
```

$$\begin{pmatrix} \{myrank < nprocs/2, (myrank+2)\%nprocs, 4000\}, \\ \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank-2)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank-4)\%nprocs, 4000\} \end{pmatrix}$$

```
29  }
30  if (x > NO_ WORK) {
31    for(i=0; i<1000; i++) {
```

$$\begin{pmatrix} \{myrank < nprocs/2, (myrank+1)\%nprocs, 4000\}, \\ \{myrank < nprocs/2, (myrank+2)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank-1)\%nprocs, 4000\}, \\ \{myrank >= nprocs/2, (myrank-2)\%nprocs, 4000\} \end{pmatrix}$$

```
39    }
40  }
```

Step 4. Run line 14. We are able to unwrap all the loops in this example. And what we get is as below.

P_1
$$\begin{pmatrix} \{myrank < nprocs/2, (myrank+5)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+10)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-5)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-10)\%nprocs, 4000000\} \end{pmatrix}$$

P_2
$$\begin{pmatrix} \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+8)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-4)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-8)\%nprocs, 4000000\} \end{pmatrix}$$

P_3
$$\begin{pmatrix} \{myrank < nprocs/2, (myrank+2)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-2)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-4)\%nprocs, 4000000\} \end{pmatrix}$$

```
30  if (x > NO_WORK) {
```

P_4'
$$\begin{pmatrix} \{myrank < nprocs/2, (myrank+1)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+2)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-1)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-2)\%nprocs, 4000000\} \end{pmatrix}$$

```
40  }
```

Step 5. Run line 15~17. The phase P_4' in the branch is changed to P_4 by absorbing the branch condition. Note that because P_4 contains runtime values. It is a persistent phase and cannot be further resolved until runtime.

$$P_1 \begin{pmatrix} \{myrank < nprocs/2, (myrank+5)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+10)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-5)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-10)\%nprocs, 4000000\} \end{pmatrix}$$

$$P_2 \begin{pmatrix} \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+8)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-4)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-8)\%nprocs, 4000000\} \end{pmatrix}$$

$$P_3 \begin{pmatrix} \{myrank < nprocs/2, (myrank+2)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-2)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-4)\%nprocs, 4000000\} \end{pmatrix}$$

$$P_4 \begin{pmatrix} \{x > NO\_WORK \;\&\&\; myrank < nprocs/2, (myrank+1)\%nprocs, 4000000\}, \\ \{x > NO\_WORK \;\&\&\; myrank < nprocs/2, (myrank+2)\%nprocs, 4000000\}, \\ \{x > NO\_WORK \;\&\&\; myrank >= nprocs/2, (myrank-1)\%nprocs, 4000000\}, \\ \{x > NO\_WORK \;\&\&\; myrank >= nprocs/2, (myrank-2)\%nprocs, 4000000\} \end{pmatrix}$$

Step 6. Run line 18~20. P_1 is adjacent to P_2. But the P_1 and P_2 merged result has a communication degree of 4 and CanBeMerged will return FALSE. P_2 is adjacent to P_3, and the resulting communication degree is 3 and can still be fit into the network. Thus they are merged as phase P_5. Because P_4 contains dynamic information, we do not try to merge P_5 and P_4.

$$P_5 \begin{pmatrix} \{myrank < nprocs/2, (myrank+2)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+8)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-2)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-4)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-8)\%nprocs, 4000000\}, \end{pmatrix}$$

Step 7. We reach line 21. There are no phases that can be merged, and there are no loops. The phase partition algorithm finishes. The final communication pattern contains 3 phases.

$$P_1 \begin{pmatrix} \{myrank < nprocs/2, (myrank+5)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+10)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-5)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-10)\%nprocs, 4000000\} \end{pmatrix}$$

$$P_5 \begin{pmatrix} \{myrank < nprocs/2, (myrank+2)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+8)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-2)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-4)\%nprocs, 4000000\}, \\ \{myrank >= nprocs/2, (myrank-8)\%nprocs, 4000000\}, \end{pmatrix}$$

$$P_4 \begin{pmatrix} \{x > NO\_WORK \,\&\&\, myrank < nprocs/2, (myrank+1)\%nprocs, 4000000\}, \\ \{x > NO\_WORK \,\&\&\, myrank < nprocs/2, (myrank+2)\%nprocs, 4000000\}, \\ \{x > NO\_WORK \,\&\&\, myrank >= nprocs/2, (myrank-1)\%nprocs, 4000000\}, \\ \{x > NO\_WORK \,\&\&\, myrank >= nprocs/2, (myrank-2)\%nprocs, 4000000\} \end{pmatrix}$$

END.

In the above case study, we assume that the network capacity is 3 and thus we had no opportunity to see the effect of FilterUntilFit(). Below we will redo the same case study with network capacity of 2. Note that both 3 and 2 are not realistic network configurations. We pick them to demonstrate our phase partitioning algorithm. When *NET* is 2, the 1st two steps remain unchanged. We resume from step 3 and mark it as step 3B.

Step 3B. Run line 7~12. The algorithm in line 8~11 produces the same results as stem 3. Since the network capacity is 2, the results cannot fit in the network. Thus the FilterUntilFit() in line 12 will generate the following results. Because the communication is evenly distributed across all connections, we randomly pick two.

```
 1   for(i=0; i<1000; i++) {
       ( {myrank < nprocs/2, (myrank+5)%nprocs, 4000},
       ( {myrank < nprocs/2, (myrank+10)%nprocs, 4000},

9   }
11   for(i=0; i<1000; i++) {
       ( {myrank < nprocs/2, (myrank+4)%nprocs, 4000},
       ( {myrank < nprocs/2, (myrank+8)%nprocs, 4000},
19   }
21   for(i=0; i<1000; i++) {
       ( {myrank < nprocs/2, (myrank+2)%nprocs, 4000},
       ( {myrank < nprocs/2, (myrank+4)%nprocs, 4000},
29  }
30  if (x > NO_ WORK) {
31    for(i=0; i<1000; i++) {
       ( {myrank < nprocs/2, (myrank+1)%nprocs, 4000},
       ( {myrank < nprocs/2, (myrank+2)%nprocs, 4000},
39    }
40  }
```

Step 4B. Run line 14. We are able to unwrap all the loops in this example to get

P_1 $\begin{pmatrix} \{myrank < nprocs/2, (myrank+5)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+10)\%nprocs, 4000000\}, \end{pmatrix}$

P_2 $\begin{pmatrix} \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+8)\%nprocs, 4000000\}, \end{pmatrix}$

P_3 $\begin{pmatrix} \{myrank < nprocs/2, (myrank+2)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000000\}, \end{pmatrix}$

```
30  if (x > NO_WORK) {
```

P_4' $\begin{pmatrix} \{myrank < nprocs/2, (myrank+1)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+2)\%nprocs, 4000000\}, \end{pmatrix}$

```
40  }
```

Step 5B. Run line 15~17. Again the phase P_4' in the branch is changed to P_4 by absorbing the branch condition. P_4 contains runtime values. It is a persistent phase and cannot be further resolved until runtime.

P_1 $\begin{pmatrix} \{myrank < nprocs/2, (myrank+5)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+10)\%nprocs, 4000000\}, \end{pmatrix}$

P_2 $\begin{pmatrix} \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+8)\%nprocs, 4000000\}, \end{pmatrix}$

P_3 $\begin{pmatrix} \{myrank < nprocs/2, (myrank+2)\%nprocs, 4000000\}, \\ \{myrank < nprocs/2, (myrank+4)\%nprocs, 4000000\}, \end{pmatrix}$

P_4

$$\begin{pmatrix} \{x > NO\_WORK \;\&\&\; myrank < nprocs/2, (myrank+1)\%nprocs, 4000000\}, \\ \{x > NO\_WORK \;\&\&\; myrank < nprocs/2, (myrank+2)\%nprocs, 4000000\}, \end{pmatrix}$$

Step 6B. Run line 18~20. Since network capacity is 2 and the communication degree of each phase is already 2, no adjacent phases can be merged. No work is done in this step.

Step 7B. We reach line 21. No further change can be made to the results. The phase partition algorithm stops. The final communication pattern contains 4 phases, shown in step 5B.

END.

From the case study, we demonstrated how to apply our phase partition algorithm to compose communication phases from MPI programs step by step.

7.0 RUNTIME SCHEDULING

Runtime scheduling plays an important role in effectively utilizing the compiler identified communication patterns for achieving efficient communication. The switching architecture that we consider to evaluate this research is based on the one proposed in [21]. It uses two types of networks and requires a runtime scheduler to achieve high communication performance. In this dissertation, we augment the runtime scheduler with the capability of taking advantage of compiler assistance. With the work in this research, the runtime system is capable of handling compiler provided deterministic (for static and persistent communication) or heuristic (for dynamic communication or code artifacts like loops) network configuration instructions.

7.1 RUNTIME SYSTEM AND NETWORK SCHEDULING

The multiprocessor model targeted in this research is described in Section 3.1. A multiprocessor system with this model deploys a number of circuit switching networks. Such networks have very high bandwidth, very low latency, and relatively long connection establishment overhead—which is measured in milliseconds in optical MEMS-based circuit switches.

Based on these observations, it will be extremely inefficient if a multiprocessor system establishes the needed circuits after the communication requests arrive. To avoid such millisecond magnitude delays, a packet switching network is included in a targeted

multiprocessor system to send messages if the needed circuits are not established at communication time. To keep the cost of the proposed multiprocessor system low, the packet switching devices need to be cheap. It implies that the packet switches may not have high bandwidth and their latencies may be relatively long. In general, at runtime we dispatch a message to the circuit network only when there is a circuit which can satisfy the request; otherwise, the message is dispatched to the slow packet switching network.

In general, runtime scheduling relies on prediction. An optimal runtime scheduler is expected to be able to make all connections available at the time they are needed. To obtain efficient communication, there are two types of information to be predicted. One is when and which connections to establish in preparation for the upcoming communication requests. The other is when and which connections to evict to make room for the new ones.

In the proposed machine model, the runtime scheduler dispatches a message to the circuit switching network only when the needed circuit is available. Thus the effectiveness of utilizing the circuit networks heavily depends on the ability of the runtime scheduler to correctly predict the connections needed by the upcoming communication requests, and when the prediction can be made. The network capacity of any given multiprocessor system is limited and the runtime scheduler may not be able to implement all the needed connections simultaneously. In such scenarios, the runtime scheduler needs to predict which conflicted existing connection(s) can be torn down to establish the new ones. In short, the runtime scheduler needs to predict when to establish what connection(s), and evict what existing connection(s) to make room for the new one(s).

There are many research efforts addressing the communication prediction problem. For example, Sakr et. al. used machine learning techniques to predict multiprocessor memory access

pattern [100]. Mukherjee and Hill used an address-based predictor to predict coherence messages for shared memory [101]. Zhu implemented an aging based predictor to decide when to tear down a connection in a Time Division Multiplexing (TDM) switch design. We experimented with several sophisticated prediction techniques with the machine model described Section 3.1. For instance, one of the approaches is to predict a number of potential connections based on runtime traffic volume, then based on an adaptive threshold decide to pick one or multiple connections as the final prediction results. However all these efforts led to a surprising conclusion: the simplest straightforward approach is the best. That is a threshold-based circuit establishment strategy combined with a LRU victim-picking strategy with simple parameter tuning beats or performs as well as all other sophisticated approaches that were investigated.

Our runtime scheduling algorithm uses a threshold-based circuit establishment approach and is detailed in Figure 20. When a communication request with a volume higher than a threshold is observed, the algorithm tries to establish a connection that can satisfy the request in the circuit switching network. When the circuit switching network has no space to establish a predicted connection, the scheduler will evict one (in the cases of only one connection is using either the requested source or the destination) or two (in the cases of that one connection is using the requested source and the other one is using the requested destination) connection(s) with LRU algorithm to make room for the new one.

The reason that a simple prediction works as well as sophisticated predictions is easy to understand. The predictor in the runtime scheduler is to forecast the communication patterns within parallel applications. The communication pattern in a parallel application may be random. But when a pattern is random, all prediction algorithms have no chance of making accurate prediction. A naive predictor will do just as well as a sophisticated predictor. The well accepted

conclusion about the communication patterns of parallel applications is that most parallel applications have regular communication patterns even when the communications are dynamic [3, 7, 8, 62, 67]. Such a fact indicates that in most cases a communication pattern predictor works in a world with strong regularity. In such kind of environments, a simple predictor has the capability of producing results as well as a sophisticated predictor. That is why we found out that our very simple approach generates promising results.

```
 1 if req q <source,destination,volume> can be satisfied by circuit c then
 2   dispatch q to c
 3   return
 4 else
 5   dispatch q to the slow packet network
 6 if (q's volume v >= threshold T) then
 7   /* try to establish circuit needed by q */
 8   if these exits a circuit network CN_i whose source and destination port
are free then
 9     establish circuit<source,destination> on CN_i
10     return
11* pick up the circuit network CN_v whose source and destination ports
          have the longest idle time (sum of the idle time of both ports)
12   tear down the connection(s) using src and dst port on CN_v
13   establish circuit<src,dst> on CN_v
14 return

* Note that we keep track of the last used time for all ports in all networks.
Thus we know the idle time for each port.
```

Figure 20. Runtime Scheduler Circuit Establishment Algorithm

Note that pure runtime scheduling cannot beat compiled communication approach for applications dominated by static and persistent communications, which are the majority of parallel applications. There are two major reasons. First, compiled communication can accurately analyze static and persistent communication patterns, while pure runtime scheduling relies on predictors which may make mistakes. Second, compiled communication can identify the communication pattern and decide network configurations much earlier, say at compile or load

time, than purely runtime scheduling. A correct prediction is still useless if there is no enough time to establish the connection(s) after prediction is made. The experimental results reported in Chapter 9.0 verify the effectiveness of compiled communication compared to purely runtime prediction.

7.2 COMPILER ASISTED RUNTIME SCHEDULING

From the runtime scheduler's point of view, the compiler takes over most of the runtime prediction work with high prediction accuracy. This makes it possible to keep the runtime scheduler simple, fast, and effective. But there still exist two major challenges to conquer. First, how can the runtime scheduler effectively utilize the compiler provided knowledge to achieve efficient communication? Second, how can the compiler help the execution of applications dominated by dynamic communication? For these applications, the compiler is no longer able to provide network configuration information to the runtime scheduler. In this section we will answer these two questions.

7.2.1 Establishment of Compiler Identified Network Configuration

After the compiler identifies the communication pattern and compiles them as network configurations, it is the runtime scheduler which sets up the connections in multiprocessor systems. The runtime scheduler needs to work properly to implement efficient communication. This means that generally the scheduler should not allow a runtime predicted connection to overwhelm a compiler identified connection. The reason is obvious: the compiler identified

connection will certainly service certain amount of communication traffic; while a runtime predicted connection may actually be a part of a stable communication pattern or just a single-instance message. Therefore it has less probability to be able to actually serve a lot of traffic. So we favor compiler identified connections over runtime predicted connections. The scheduler shall establish and pin the compiler identified connections in the circuit switching network.

For each communication phase in the execution of parallel application, the runtime scheduler has two options regarding when to preload the network configuration into the circuit switching networks. One option is called global network configuration preloading; the other is referred to as local network configuration preloading. They are detailed below.

7.2.1.1 Synchronized Network Configuration Preloading

With this type of preloading, all participating nodes synchronize at the beginning of each communication phase. Once the synchronization is achieved, the scheduler loads the entire network configuration of that phase. This also implies that the network configuration of a phase can be entirely destroyed at the end of that phase.

This approach minimizes the interference between two adjacent phases. Once the network configuration is in place, we can tell that all the connections belonging to the configuration are immediately usable. Also, this allows us to schedule all the connections in the configuration offline either by the compiler if it is provided with the number of circuit networks, or by a configuration preprocessing tool or the MPI program loader. In either case, we can pick a sophisticated scheduling algorithm to build an optimal or near optimal network configuration because scheduling is offline. In our case we use the algorithm described in Section 6.2.1. The algorithm is used in both the compiler and a standalone network configuration preprocessing tool. By making the algorithm a standalone tool, we only need to re-compute the network

configuration when running the application in systems with different number of circuits switching networks, and no longer need to re-compile the application every time.

This approach will not work well if different nodes of a parallel application execution instance run at different pace and enter/leave a communication phase at very different times. But this can rarely happen. All the participating nodes of a parallel application run need to cooperate with each other to accomplish the entire task. One node cannot make any further progress until it receives the data from certain partner nodes, even if it finishes much earlier. Such implicit synchronizations hold back the faster nodes and make all the participating nodes well synchronized.

7.2.1.2 Unsynchronized Network Configuration Preloading

In a different approach, once a node enters a phase, the runtime scheduler starts the attempts to establish connections for it until all connections are established or the node leaves the phase. This approach does not enforce synchronizations at phase boundaries.

With this approach, the network configuration just specifies which connections are included, and the scheduler must pick a network at runtime to establish a specific connection. The most significant advantage of this approach is that the faster nodes can start establishing connections for a new phase earlier and tear down the connections of an ending phase earlier. Thus it may achieve an overall better actual network configuration.

Compared to synchronized preloading, this approach has some disadvantages. One of them is the runtime scheduling overhead. When using this approach, the scheduler has to pick a network to establish a specific connection. When there is no free network, the scheduler has to run the replacement algorithm trying to select one or two victim connection(s). Another problem

is that all the scheduling decisions are made locally. In general, a locally made decision cannot complete with globally made decision.

In general, there is more uncertainty and overhead with unsynchronized preloading approach. After an overall evaluation, the global preloading approach is preferred.

7.2.2 Heuristic Hints at Loop Boundaries

For parallel applications that are dominated by dynamic communications, the compiler cannot provide deterministic network configurations to the runtime scheduler. But the compiler still has the capability of providing useful heuristic hints to assist the runtime scheduler in doing a better job.

The major opportunity comes from the fact that compilers can identify loops. Parallel applications tend to have many communication phases during its lifetime. Each communication phase is very likely written as a loop. When the compiler recognizes that a loop contains communications, it properly observes a communication phase. In this case, the complier can insert a heuristic hint at the loop entrance boundary to tell the runtime scheduler that there may exist a communication phase, and insert a heuristic hint at the loop exit boundary to tell the runtime scheduler that a communication phase may have completed.

One of the potential beneficial actions the scheduler can take in response to the above limits is to tear down all existing connections at the loop exit point—we call this network configuration flush. When the next communication pattern is much different than the current one, this can help the scheduler save time on tearing down the connections established for the old communication pattern but no longer needed by the new communication pattern.

The loop entrance hints can be used to speed up communication phase transition. Each node can signal the scheduler that it enters a loop. When the scheduler has to tear down existing ones for establishing new ones, it can be more conservative to evict connections established within the loop and be more aggressive to tear down a connection established before the loop. Furthermore, when the scheduler finds that many existing connections have been evicted since entering a loop, it is very likely this is caused by that the loop having a different communication pattern than the previous phase. In this case, it can aggressively tear down all connections established before the loop. By doing this, the runtime scheduler is able to spend less time to establish a new pattern.

Of course, the hints could be inaccurate. For example, two adjacent communication phases may have very similar communication patterns that are larger than the capacity of the circuit switching networks. When this is the case, the scheduler will observe significant amount of connection evictions since entering the loop. But it cannot get the expected amount of benefits from aggressively tearing down connections established before the loop. Thus the scheduler needs to have the capability of turning on or ignoring the compiler provided hints. Certain evaluation runs can be performed for the application to identify what is a better choice—using the hints or ignoring them.

However, from both observations in this research and many published works related to communication characters of parallel applications, the same conclusion is obtained that most parallel applications have good locality and regular communication pattern.

7.3 SUMMARY

In summary, the runtime scheduler in a multiprocessor system sets up or tears down circuits based on the predictions made by the compiler or a simple predictor. Given that most parallel applications are dominated by static or persistent communication, compiled communication can usually provide accurate network configuration information to the runtime scheduler. Runtime scheduler should be able to effectively utilize the compiler identified network configurations. Otherwise, we cannot achieve the value of the compiler work. The experimental results presented in Chapter 9.0 show that the runtime scheduler developed in this research can effectively utilizes the knowledge provided by the compiler.

8.0 IMPLEMENTATION OF THE FRAMEWORK AND SIMULATOR

To demonstrate the effectiveness of our compiler techniques, we implemented an experimental compiler. The compiler implements the compilation framework described in Figure 4. As we have described, for cost and time limits, it is infeasible to build a real circuit switching enabled multiprocessor system in the context of this research. Thus we implemented a simulator to emulate our targeted multiprocessor systems. We use the simulation results to show the effectiveness of our compiler techniques.

In this section, we describe the implementation details for our experimental compiler. Then, we present the implementation of the multiprocessor simulator.

8.1 IMPLEMENTATION OF EXPERIMENTAL COMPILER

We build our experiment compiler on top of the SUIF compiler infrastructure [79]. The architecture of the compiler is shown in Figure 21. The SUIF compiler infrastructure is an open source, source to source compiler research toolkit supporting both C and Fortran 77 languages. The front end of the SUIF compiler compiles parallel applications to SUIF intermediate format. A tool, porky, from the SUIF compiler system is used to perform basic transformations, such as copy propagation and constant propagation in order to discover more details about the characteristics of different communication (e.g. static, dynamic, or persistent communications).

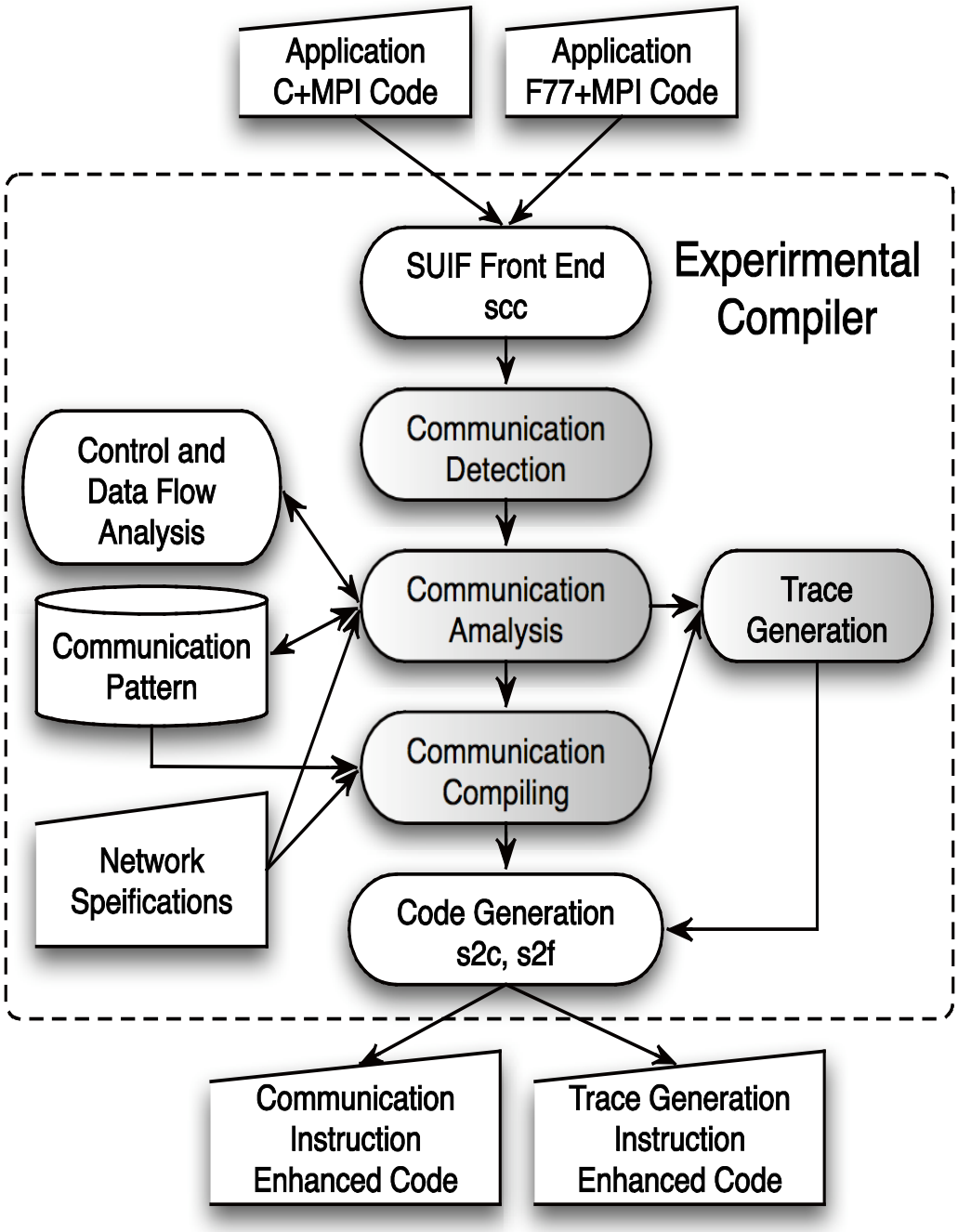


Figure 21. Structure of the Experimental Compiler

The different components described in our compilation framework are implemented as different passes. Each pass works on the SUIF intermediate representation and different passes can share information by using the SUIF annotation mechanism. Compiler techniques developed in this research are used in different passes. In the following sections, we describe the details of each pass in our experimental compiler.

8.1.1 Communication Detection Pass

In this pass, we identify all the MPI communication calls and the corresponding parameters for each MPI function. This is achieved by writing a pass to analyze the SUIF intermediate code. Within the intermediate code, we can identify all function calls, the name of the function and the parameters of the function.

According to the MPI standards, we know the function name and synopsis of the legal MPI functions. Based on this knowledge, we can identify, within the compiler, the MPI operations that occur in an application. Note that the implemented experimental compiler does not recognize all MPI functions since many of them are rarely used. We implemented the experimental compiler in such a way that it can identify a set of frequently used MPI functions which is a superset of the MPI APIs that appeared in the benchmark programs investigated in this research. To make the compiler works as if it knows all legal MPI functions, we implemented a discover-and-alarm mechanism—the experimental compiler will abort and ask for special handling when it identifies a function which looks like a new MPI function. This way we can avoid spending a huge amount of time to handle the MPI functions (there are hundreds of them) that are rarely used in programming practice.

The fundamental information about all MPI operations within a MPI application is identified in this pass.

8.1.2 Communication Analysis Pass

During communication analysis pass, the MPI operations and parameters used in the MPI functions identified during the communication detection pass are used to compose communication patterns. Control and data flow analysis and inter-procedural analysis are used to classify the type of each communication operation. If the source, destination, and message volume of a communication operation can be resolved as constants, the communication is static.

We extend inter and intra-procedural constant propagation into symbolic expression propagation to determine whether communications are static. Thus if a communication operation is discovered to be persistent (as described in Figure 6 (b)) in the early stages, it still needs to be further analyzed to check if it is actually static. For example, it is necessary to recognize and consider the processor ID, number of processors, problem size, and various other parameters as static to ensure that the topology can be determined statically.

As described before, we use loops as basic phase delimiters. We always try to merge contiguous phases to form larger phases when the phase can still fit into the underlying network. To obtain proper communication phases within the code, we use the information discovered in CDFG analysis and communication detection pass. The proper determination of phases is based on factors such as phase length, amount of static and/or persistent communication present, and communication to computation ratio. Phase manipulation operations such as Merge, Filter and

Unwrap are implemented in our experimental compiler and used in both the communication analysis pass and the communication compiling pass.

Our implementation assumes that a multiprocessor system includes *K* circuit switching networks where *K* is a configurable parameter. Each network is a crossbar capable of establishing an arbitrary permutation between the sources and destinations. Other types of networks may be implemented and added to the compiler for use.

8.1.3 Communication Compiling Pass

The communication compiling pass further optimizes the communication pattern identified during analysis, compiles the pattern to network configuration directives and inserts them into the application to assist runtime scheduler configuring the interconnections. The communication compiling pass exposes static and persistent communications to circuit switching interconnection networks with the goal of reducing the communication overhead. For example, this experimental compiler inserts instructions to preload network configurations for the switches proposed in [21]. Currently, two types of network configuration instructions will be inserted to parallel applications by the experimental compiler.

Network configuration setup instructions are used to pre-establish network connections. It can hide the setup overhead of connections and overlap network control with computation.

Network configuration flush instructions are used to flush the current network configuration or a subset of circuits from the current configuration. Such instructions can be used to remove the expired circuits. It provides potential to speedup the parallel applications as it reduces contention for the circuits.

For static communication patterns, the content of the communication pattern derived during analysis is inserted into the code using network configuration setup instructions designed to pre-configure the network at the beginning of each communication phase. For persistent communication patterns, the communication analysis component provides communication pattern information as a function of variables that will not be known until run-time. Code for calculating these symbolic expressions at the appropriate locations after all the required parameters are known is generated and inserted into the code. The definition scope and initialization of variables in the symbolic expressions, in addition to the location of the communication operations, put constraints on the locations where these instructions may be inserted.

8.1.4 Tracing Generation Pass

The experimental compiler also has the capability to automatically add trace generation instructions (e.g. print statements) within MPI applications. This pass relies on the information discovered in the communication detection pass and the communication analysis because it generates traces for both MPI functions and related code structures such as branches, loops and procedure boundaries.

This is achieved by augmenting the original SUIF intermediate code with print instructions. The inserted print instructions are able to print out all the information we just described in the previous paragraph. Note that the resulting instrumented SUIF intermediate code can be compiled and executed on the parallel platform just like the original code. The only difference is that the instrumented code includes print instructions that generate traces.

The generated traces are primarily used to study communication patterns of an execution instance and/or verify that the communication patterns detected by the compiler are accurate. The approach used here for trace generation is an instrumentation approach. This means that we cannot completely prevent the trace generation code from disturbing the application execution. To relieve the impact, we design the trace generation code carefully. For example, the execution time that a communication operation needed at run-time heavily depends on the particular communication network.

However, our goal is to study the performance of an application on a multiprocessor system whose interconnection network is re-configurable. Thus, in the trace for simulation, we record the time between two adjacent communication operations. In this way, the performance of the network on which we collect traces will not impact our simulation. Note that even without considering compiled communication techniques, this pass allows the compiler to be a useful tool for automating trace generation for MPI applications and relieve researchers from the trouble of generating traces manually.

8.2 IMPLEMENTAION OF MULTIPROCESSOR SIMULATOR

An event-driven multiprocessor system simulator was developed for performance study. It was developed using C++ with CSIM. CSIM is a process-oriented discrete-event simulation package for use with C or C++. While developing a simulation program with CSIM, the simulated system is typically modeled as a collection of CSIM processes which can interact with each other. For instance, entities in a multiprocessor system, such as processors, NICs, circuit switching scheduler, are represented as CSIM processes. A simulation program, or simulator, developed

with CSIM can produce estimations of time and performance of the modeled system. Simulators developed with CSIM can yield insight into the run-time behavior of the modeled system. Therefore, CSIM is a reasonable choice for the development of the multiprocessor system simulator for this research.

In brief, we simulate multiprocessor systems that follow the machine model described in Figure 3. Each simulated system contains N processors and its diagram is shown in Figure 22.

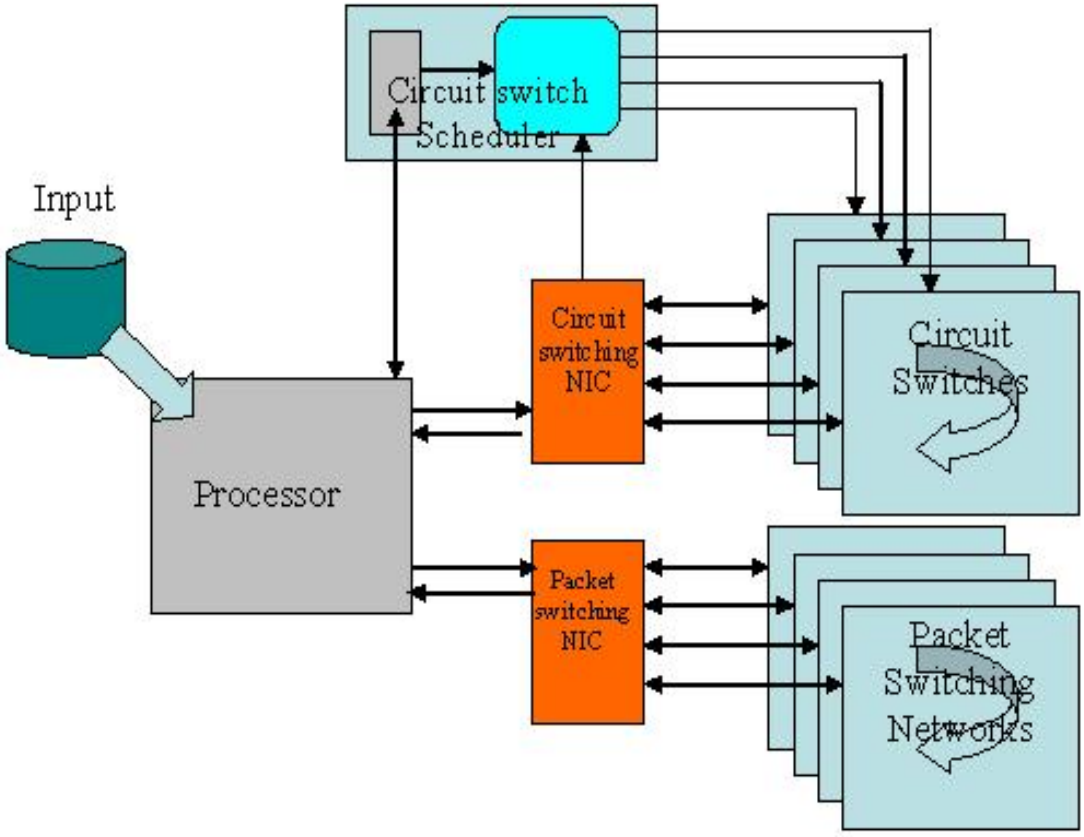


Figure 22. Paradigm of the Simulated Multiprocessor System

In the simulated multiprocessor system, each processor has a circuit switching NIC connected to a configurable number of circuit switching networks and a packet switching NIC connected to a configurable number of packet switching networks. Hence, a simulated system

may have a configurable number of circuit switching networks and a configurable number of packet switching networks simultaneously. This design of the multiprocessor system simulator provides us the flexibility of configuring different kinds of multiprocessor systems for experiments and performance study. In general, given a particular configured multiprocessor system, circuit switching networks are used to accommodate static and persistent communications that are typically dominant in parallel applications; and packet switching networks are used to accommodate collectives and short-lived dynamic communications. Separating the communication classes, as proposed in this research, enables us to target each class of communication with the most appropriate network technology and operating mechanism. Specifically, static and persistent communications will be scheduled and dispatched to the circuit switching interconnect network at compile-time.

8.2.1 Processing Elements

Typically, a multiprocessor system contains a number of processing elements, or processors, which are interconnected by certain interconnection networks. Hence, processors are one kind of basic entities that a multiprocessor system simulator needs to simulate. In a simulated multiprocessor systems, each processor reads simulation commands from an input file. Each input file contains a sequence of simulation commands which describe the run-time program behaviors of a processor during a parallel application execution. Note that the purpose of this simulator is to study the communication behaviors of parallel applications on multiprocessor system. Therefore, we focus on modeling communication operations in parallel applications. Each communication operation is abstracted as a communication command. In the trace files communication commands can be separated by computation commands “COMP t”, each of

which emulates a serial of computation over time *t*. This is achieved by taking timestamps right before and after each communication operation. The time *t* of a computation command is calculated by the timestamp right after a communication operation minus the timestamp right before the next communication operation.

The frequently used simulation commands are listed in Table 4. They can emulate typical communication operations in real parallel applications execution. When a simulation runs, the commands in each input file are read and emulated by each corresponding processor one by one. In general, the simulation commands can be classified in two categories, point-to-point communication commands and collective communication commands. Point-to-point communications include send and receive. There are a number of different send commands defined in the MPI standard. But for simplicity, they can be modeled into two classes of send commands, blocking send—SEND, and non-blocking send—ISEND. Blocking send returns when the message leaves the buffer of the NIC. Non-blocking send returns control as soon as the message header is stored in the buffer of the NIC.

Similarly, there are two types of receive commands, blocking receive—RECV, and non-blocking receive—IRECV. Blocking receive returns control when the matching message is actually received by the processor. Non-blocking receive returns control without actually receiving the message. For each non-blocking receive we usually has to do a WAIT operation later to guarantee that the expected message eventually arrives.

There are a few collective communications. The basic one is the barrier. Most collective communications have an implicit barrier at the beginning. For the purpose of this research, the simulator emphasizes the point-to-point communications and will not simulate the details of collective communications, but only emulate the functionalities with a configurable time delay.

Table 4. Simulation Commands

Commands	*Syntax*
SEND dest bytes tag comm.	Blocking send operation. It sends "bytes" bytes to the destination processor "dest" in communicator "comm" and the message is labeled with tag $tag
ISEND dest bytes tag req comm.	Non-blocking send operation. It sends "bytes" bytes to the destination processor "dest" in communicator "comm.", the message is labeled with tag "tag". This operation returns a handle "req".
RECV src bytes tag comm.	Blocking receiving operation. It tries to receive "bytes" bytes from the source processor "src" and the message should be labeled with tag "tag"
IRECV src bytes tag req comm.	Non-blocking receiving operation. It tries to receive "bytes" bytes from the source processor "src" and the message should be labeled with tag "tag". This operation returns a handle "req".
BARRIER comm.	Simulate a barrier operation among all involved processors in communicator "comm".
BCAST bytes comm root	Simulate a broadcast operation in communicator "comm". The root processor is "root" and data size is "bytes".
REDUCE bytes comm root	Simulate a reduce operation in communicator "comm". The root processor is "root" and data size is "bytes".
WAIT req	Simulate a wait operation. It returns after operation "req" finishes.
COMP t	Emulate a serial of computation over time "t".
NETSET net_config_id	Load the network configuration (a number of connections) with net_configration_id from a provided network configuration file.
NETFLUSH net_config_id	Tear down the connections specified by the net_config_id.

The COMP command is used to model the computation in the program. No details of computation are simulated. The computation is simulated by holding for the specified amount of time in CSIM. The amount of time depends on the original amount of time from trace files and the simulated CPU frequency.

Typically, the traces are provided as input files. The trace files are obtained from the actual execution of an application on a multiprocessor system. The application contains instrumented instruction to print the traces of MPI operations and related code artifacts, such as loop begins and ends. The instrumentation is done automatically by the trace generation pass of the research compiler. Of course, we can also directly compose trace files with simulation commands to simulate a particular synthetic program scenario. In order to be able to study the feasibility of some of network configuration strategies (such as what is the proper opportunity to tear down a connection or keep a connection for how long) in typical regular communication patterns—such as 2-D and 3-D mesh or totus traffic—besides those we discovered from real benchmarks, we also augmented our simulator to be able to automatically generate these regular traffic patterns to simulate. Note that when generating traces for the simulator, the trace generator is responsible for converting ranks within communicators other than MPI_COMM_WORLD to ranks within MPI_COMM_WORLD.

Note that each processor connects to a circuit switching NIC and a packet switching NIC. Each NIC, either the circuit switching NIC or the packet switching NIC, is connected to a configurable number of ports. Each port connects to an independent packet switching network or circuit switching network accordingly. Each NIC has one sending buffer and one receiving buffer for each port.

The NIC monitors the sending buffer for incoming (from the PE) messages and dispatches them when new ones are put to the buffer. Receiving buffers work in passive mode. It is the sender and other entities on the data path who move a packet/message forward and finally store it at the receiving buffer at the destination. The receiver PE retrieves the message from the receiving buffer. Different types of NICs process messages differently. A circuit switching NIC sends messages directly to the receiving circuit switching NIC through a circuit switching network, while an electronic NIC breaks each message into multiple packets and send each packet to a packet switching network.

8.2.2 Packet Switching Networks

In our simulated systems, in addition to circuit switching interconnection networks, packet switching interconnection networks are also supported. There are two reasons to also include packet switching networks in this research.

First, we need a very fast packet switching technique to server as our performance comparison target. The goal of this research is to develop a cost-efficient solution for tomorrow's multiprocessor systems. Specifically, we expect to achieve the same level, or even higher level of communication performance than next generation packet switching networks using much less expensive circuit switches. Thus the packet switching technique to which we compare must be very competitive for the future interconnection network design. We chose fully buffered crossbar switches (BCS).

It has been recognized that switches which deploy buffers at the cross-point can obtain very promising performance [46, 73]. However, BCS design can increase the cost of the switches extremely high and make it infeasible in practice. Note that we elected simulation based

approach in this research. It relieves us from the pressure of actual cost for packet/wormhole switching devices. Therefore choosing a more powerful design, switch-wide fully buffered crossbar, becomes a reasonable and practicable decision in this research. The advantages include: 1) it can provide very competitive communication performance even in the future; 2) it is based on, but beyond realistic systems that have been actually realized, i.e. [46,73]; 3) its architecture is simple and the verification and validation of the simulator becomes simpler and easier.

Second, we need to avoid the millisecond level circuit establishment delay whenever possible. Even though static and persistent communications are dominant in many parallel applications, there exists a small portion of dynamic communications in some applications. If we only have circuit switching networks in our targeted systems, this small portion of dynamic communications may eventually introduce significantly degradation on the overall communication performance because if the very large circuit establishment delay (e.g. in ms for MEMS-based circuit switching devices). Hence, it is worthwhile to deploy a packet/wormhole switching network to accomplish the small portion of short and dynamic communication. By including such a packet/wormhole network in our targeted systems, we effectively eliminate the potential performance degradation that may be introduced by the small portion of dynamic communication. Of course, at the same time we have to keep the overall system cost low.

To leverage the above two requests, we have to include packet switching networks in this research. However, we do not need to simulate two types of packet switching networks. We can only implement BCS networks and restrict its performance to emulate the behavior of a cheap slow packet switching network.

Note that to avoid the huge circuit establishment overhead, we always deploy one and only one implicit slow packet/wormhole switching network in a circuit switching enabled multiprocessor system.

The simulated BCS packet switch design is detailed below.

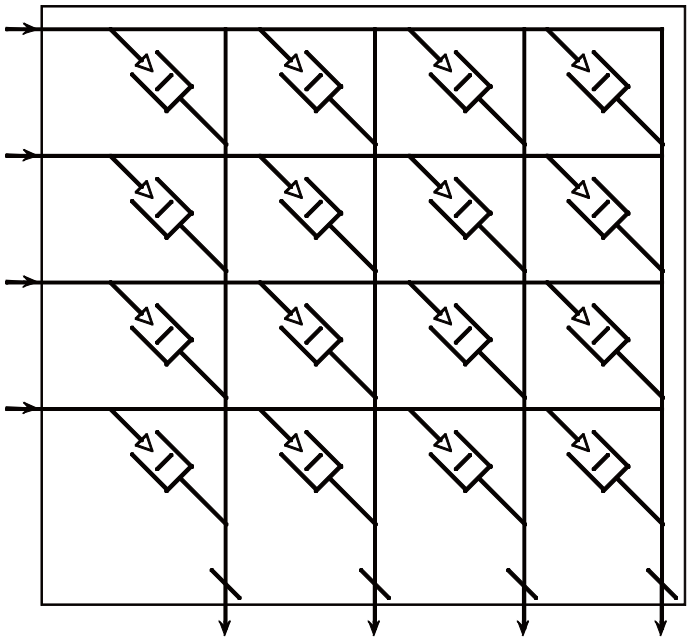


Figure 23. 4 × 4 Buffered Cross-point Switch

Each packet switching network is implemented as a 2-stage FAT tree network built using very fast fully buffered crossbar switches (BCS) [46, 73]. A BCS provides buffers at each cross-point of the crossbar within the switch. Each output port arbitrates among all its cross-points to decide which packet to deliver. Figure 23 shows an example of a 4 × 4 buffered crossbar switch which can be used as a 2 × 2 bi-directional switch. For each input/output port pair, there is a cross-point to store the packet from the connected input port and will use the connected output

port to deliver it to the next stage. All the switches that are deployed to build the FAT tree are used as bi-directional switches. And thus the FAT tree is also bi-directional.

When an N × N BCS is used as a bi-directional switch, half, N/2, of the input ports are used to receive packets from lower level switches and the rest are used to receive packets from higher level switches. Similarly, half, N/2, of the output ports are used to send packets upwards and the rest are used to send packets downwards. There is one output arbiter associated with each output port to control which cross-point in the row connected to the output port can obtain the output port for sending data out.

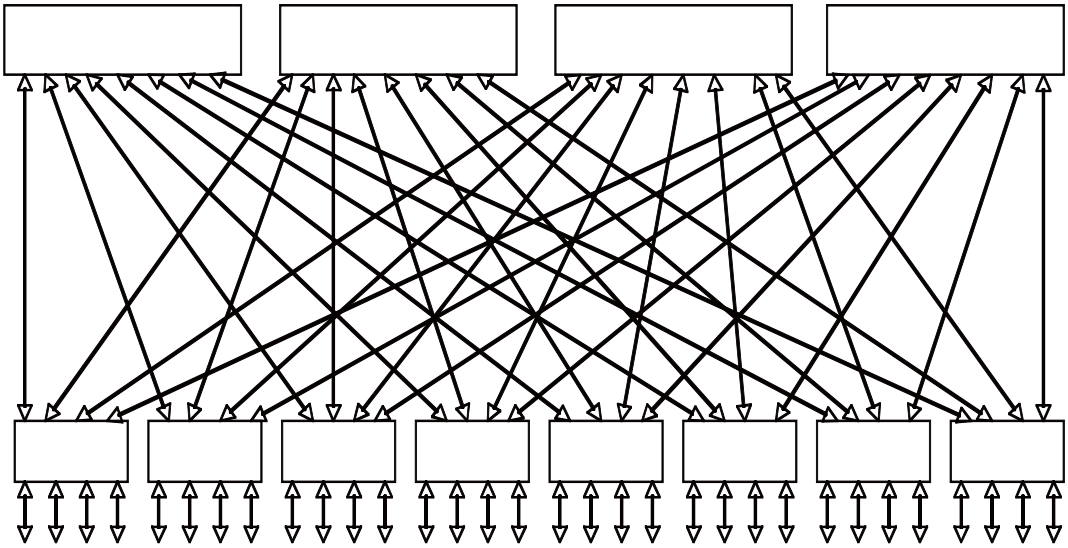


Figure 24. 32-port 2-stage FAT tree network

Figure 24 shows an example of 32-port 2-stage FAT tree network built from 8 × 8 BCS (which are used as 4 bi-directional switches). There are 8 switches that are used in the lower level and 4 switches that are used in the upper level. A larger FAT tree can be implemented by using switch elements with more ports or using more switches in each stage. The FAT tree topology can be generalized to support different topologies and different number of ports.

Note that each message will be broken into multiple small fixed-size packets when it is delivered by a packet switching network. Each packet is routed independently through the switches in the network. Thus the packets of the same message may be routed through different path to arrive the destination, and often the packets of the same message arrive at the destination out of order. They needs to be held in the destination buffer before the whole message can be properly assembled. This further increases the requirements of buffers and the hardware cost.

When simulating multiple packet switching networks, we assume that the packet switching NIC associated with a processor contains independent buffers for each packet switching network. The messages in each buffer are handled sequentially; however, the NIC can handle different buffers simultaneously. Each incoming message from the processor to the NIC is assigned a buffer randomly. But once the message is assigned to a buffer, all packets of that message can only be delivered through the corresponding packet switching network. This design reduces the mixture of packets of different messages and thus can decrease the buffer requirements.

8.2.3 Circuit Switching Networks and Scheduler

Connections in circuit switching network are dedicated direct pipes. Each pipe connects a pair of end nodes. Hence, we model each circuit switching network in the simulated multiprocessor system as an N×N switch and can be configured to realize any permutation at any time. In each simulated multiprocessor system, it is the runtime scheduler that sets up connections, mostly identified by the compiler, in the circuit switching network. The scheduler also accommodates the small amount of dynamic communications if any. The runtime scheduler is a centralized scheduler. To avoid the long connection establishment delay, a processor sends a message to the

circuit switching NIC only when there is a connection available in any of the circuit switches. Otherwise, the message is dispatched to the packet switching networks. Hence there must be at least one packet switching network in the system.

Note that there is only one centralized circuit switching scheduler which controls the establishment and destruction of dedicated connections on all circuit switching switches in the multiprocessor system. The scheduler can receive and process connection requests from any circuit switching NIC. The procedure of processing one connection is detailed below.

First, the scheduler checks whether a connection which can satisfy such a request is currently available or is in the middle of being established on one of the switches. This is possible because the establishment delay of a connection is long and the scheduler may have started establishing a connection on the same source/destination pair based on the demand of previous request(s). If there exists such a connection, either established or in the establishing procedure, the scheduler's job on this request is done.

Second, if there does not exist a connection that can satisfy the request, the scheduler will try to establish one. It examines all the circuit switching networks and try to pick one on which both the source and the destination ports are available. Then the scheduler will establish the requested connection on the selected switch and the request is satisfied.

In the third case, the scheduler cannot find a switch to establish the requested connection. The scheduler will try to tear down one or two connections on one of the switches that are less critical than the requested connection and establish the new one. In some cases, the scheduler may not be able to find any connection to tear down because both the source port and the destination port are used by connections that are more critical than the requested one. In this situation, the request cannot be satisfied. This is acceptable in this dual-network system, because

the packet switching networks can always be used to deliver messages between an arbitrary pair of nodes.

Note that the scheduler has to check the status of a connection before it can tear the connection down. Only when the connection is not currently in-use and there is no pending message in the sending buffer of the port, the scheduler can tear it down for establishing a new connection. A replacement policy is needed to decide what connections can be replaced when replacement is needed. A simple method is to replace those connections that are less recently used (LRU). The reason is simple, the connection that is least recently used will with high probability be also used less in the near future. The traffic through the replaced connections will be served by the packet switching network and when new request ask for it again, a circuit switching connection can be re-established if possible. A more elaborate policy may also take into consideration the traffic volume, and usage patterns of a connection, and other factors. However, research practice done in our group shows that our simple strategy has similar performance as many sophisticate ones.

As we have discussed in Chapter 7.0 , the runtime scheduler plays an important role to implement the performance gains enabled by adopting compiled communication techniques. It is the scheduler who is responsible for preloading, setting up, or tearing down compiler computed network configurations. It provides us the capability of more effectively taking advantage of the compiler analyzed communication patterns. When a parallel application is not statically analyzable or it contains certain amount of dynamic communications, this runtime scheduler will rely on the runtime prediction approach, presented in Figure 20, to configure the circuit networks.

8.2.4 Multiprocessor System Configuration

The simulator can simulate many system configurations by taking certain parameters from command line and/or configuration file. This provides us great flexibility of simulating different multiprocessor systems.

First of all, the simulator allows the simulated multiprocessor system to be configured to include a configurable number of circuit switching networks. That is achieved by specifying certain command-line simulation parameters. The goal of this research is to enable circuit switching in multiprocessor systems for efficient communication and accomplish competitive performance with the assistance of compiled communication techniques compared to their packet/wormhole switching counterparts.

Second, a multiprocessor system can be configured to include a configurable number of packet switching networks. This is to provide fair comparison between circuit switching with compiled communication techniques and next generation packet switching only systems. For fairness, multiprocessor systems using only packet switching networks are configured with the same amount of bandwidth as their circuit switching counterparts when we compare a pure packet-switched multiprocessor system and a pure circuit-switched system.

Further, for a given system paradigm, the simulator allows the specification of many system parameters. This gives us the flexibility of using the same simulator to emulate systems with much different communication performance. We can simulate future high performance switches with one set of parameter values and simulate switches which can only deliver ordinary performance with another set of parameter values. This relieves us from the necessity of developing two different kinds of packet/wormhole switching switches. The high performance packet switches is needed to emulate future packet-switched multiprocessor system. Those

systems will be used to validate that our circuit switching system can achieve the same or better performance. The ordinary packet switching devices is needed to deliver the dynamic communication in our circuit-switched multiprocessor system. The major configurable system parameters are listed below:

1. Number of processors
2. Processor operating frequency
3. Overhead of MPI_Send or MPI_Recv
4. Link bandwidth
5. Per-switching arbitration time
6. Minimal propagation time using only lower level FAT tree stage
7. Minimal propagation time using both level FAT tree stage
8. Link bandwidth in the circuit switching networks
9. Propagation delay in the circuit switching networks
10. Connection establishment delay in the circuit switching networks
11. Circuit replacement threshold in the circuit switching networks

Note that 4 ~ 7 are parameters for packet switching networks and 8 ~ 11 are parameters for circuit switching networks. By providing different values to these parameters, we can simulate the behavior for many multiprocessor systems.

9.0 EXPERIMENTAL RESULTS

We experimentally demonstrate the effectiveness of our approach. In this section, we first show that the compiler techniques developed in this research can identify the communication patterns accurately and efficiently. We then show the overall performance benefits when the identified communication patterns are compiled into the applications and connections are pre-established before the communication operations.

9.1 IDENTIFYING COMMUNICATION PATTERNS

To show the capability of our compiler to identify communication patterns, we consider the IS, LU, MG and CG benchmarks from the NAS parallel benchmark suite and one complete application LBMHD (Lattice Boltzmann model of magneto-hydrodynamics [58, 54]). When compiling the NAS parallel benchmarks, the total number of processors, referred to as N, has to be set as a build parameter so that it is known at compile-time.

IS: The compiler identified communication pattern for IS are shown in Table 1 and Figure 8. The weights in c−vectors and p−matrices are binary values. In phases 0 and 1, collective operations AR, AA, and AV are executed and there is no point-to-point communication. Phase 2 combines the RD collective operation with the point-to-point matrix shown in Figure 8. By using Merge and Unwrap operations, phase 0 and 1 can be combined.

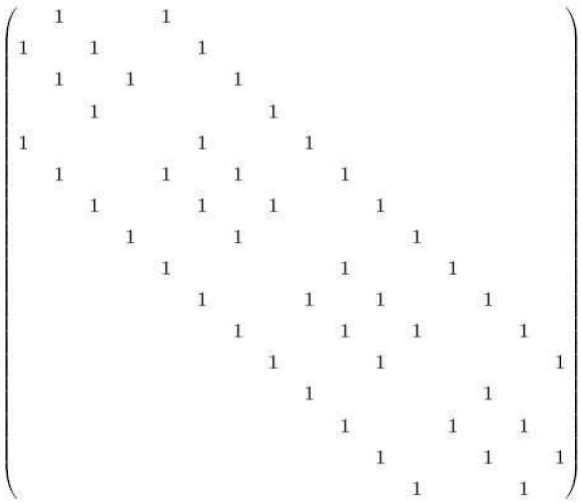


Figure 25. Single Bit p−matrix PM_LU (N = 16)

LU: The LU benchmark is demonstrated as another example. When identifying the communication patterns for LU, we ignored the collective communications from all the phases because they are trivial in this benchmark. Initially the compiler identified 11 phases. Ten of them contain only a small number of connections. Using the Merge operation the p−matrix shown in Figure 25 is obtained. From this matrix we can see that the number of destinations varies from 2 to 4 yielding a reasonably sized working set.

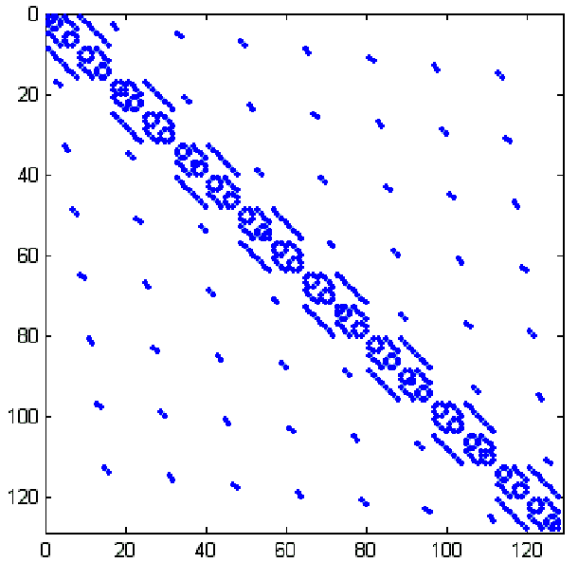


Figure 26. The *p-matrix* of CG (N = 128)

CG: The compiler initially identified two communication phases from the source code. It turns out that each of these phases is identical and can be merged. The p−matrix for the compiler predicted communication pattern is shown in Figure 26.

MG: When analyzing MG, the compiler discovered that the communication destinations depend on run-time input data. However, after the topologies are determined, they are used for an extended period of time. Hence the communication pattern is persistent. Therefore, it is only possible to construct symbolic *p−matrices* or formula lists from the source code. These *p−matrices* are resolved at runtime to configure the network.

We show the resolved 12 p−matrices for the default input file supplied with the benchmark shown in Figure 27. p−matrices 0, 1, 6-10 correspond to Figure 27 (a), p−matrices 2 and 11 correspond to Figure 27(b), p−matrix 3 corresponds to Figure 27 (c), and p−matrix 4 corresponds to Figure 27(d). p−matrix 5 is empty because the branches contain no actual point-to-point communications.

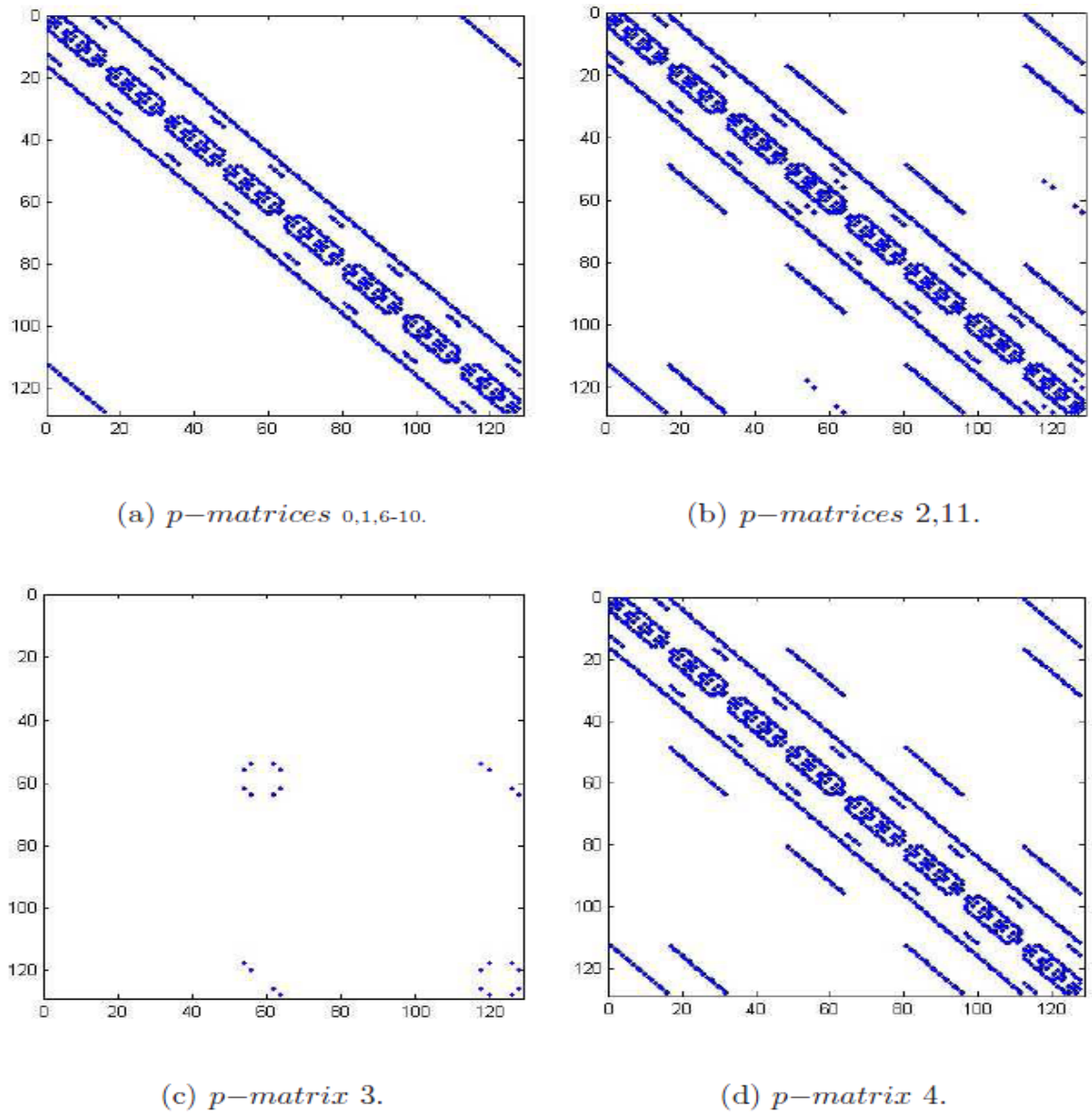


Figure 27. The *p-matrices* of MG (N = 128)

LBMHD: The communication pattern of application LBMHD is shown in Figure 28 and Figure 29. Our compiler identified a single communication phase. Because the number of processors N and the processor rank are determined at run-time and not known at compile-time, we obtain a symbolic p-matrix in Figure 28 to describe the communication pattern. Each processor has a set of four other processors as communication partners. Since *N* and *rank* are the

only symbols in the expression, this matrix is considered statically known as it may be entirely resolved at application load-time. Thus it is possible to entirely configure the network for LBMHD based on compiler analysis prior to execution. Figure 29 shows the pattern for a run on 64 processors.

$$\begin{pmatrix} ((\lfloor rank/N_y \rfloor + 1) \mod N_x + (rank - \lfloor rank/N_y \rfloor * N_y) \\ ((\lfloor rank/N_y \rfloor - 1) \mod N_x + (rank - \lfloor rank/N_y \rfloor * N_y) \\ \lfloor rank/N_y \rfloor * N_y + ((rank - \lfloor rank/N_y \rfloor * N_y) + 1) \mod N_y \\ \lfloor rank/N_y \rfloor * N_y + ((rank - \lfloor rank/N_y \rfloor * N_y) - 1) \mod N_y \end{pmatrix}$$

Figure 28. LBMHD p−matrix Described by a Formula List Where N = Nx * Ny

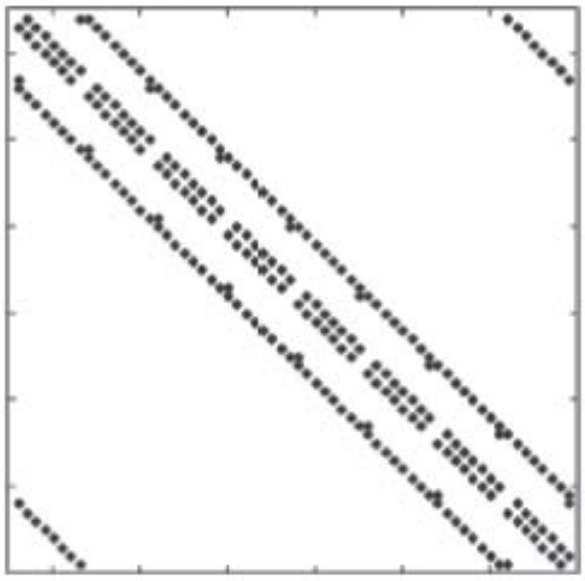


Figure 29. LBMHD p−matrix (N = 64)

The above compiler-predicted communication patterns have been verified by comparison to their counterparts extracted from traces. The communication patterns identified from the corresponding traces are identical to what the compiler has identified from the source code.

Note that we obtain the traces with the assistance of the experimental compiler. With the trace generation pass enabled, the compiler instruments each program with trace generation instructions. The instrumented program is compiled and run on multiprocessor system normally to generate trace files.

9.2 PERFORMANCE ANALYSIS

To demonstrate the effectiveness of the techniques developed in this research, we perform performance analysis with the experimental compiler and multiprocessor system simulator, which are detailed in Chapter 8.0 .

Our experimental methodology is presented in Figure 30. First, we apply the experimental compiler to the original source code of parallel applications. The compiler detects the communication patterns of the applications, compile them as network configurations, and inserts network configuration directives into the application for preloading network configurations. The experimental compiler also enhances the programs with trace generation instructions. Then source code is re-generated with enhanced trace generation instructions and network configuration instructions. The enhanced source code is fed to the real parallel computing system compilers for compilation and execution. The traces of MPI programs are collected on the real parallel computing systems, such as clusters or supercomputers. Besides communication operations and computing commands, the traces include network configuration directives. These traces along with network configurations generated from communication patterns are fed into the simulator. Statistics are collected during simulations and are used to study performance.

We focus on investigating the average message delay of the benchmark programs. This is because that it is the most meaningful performance metric to demonstrate the performance of communication subsystems in multiprocessor systems.

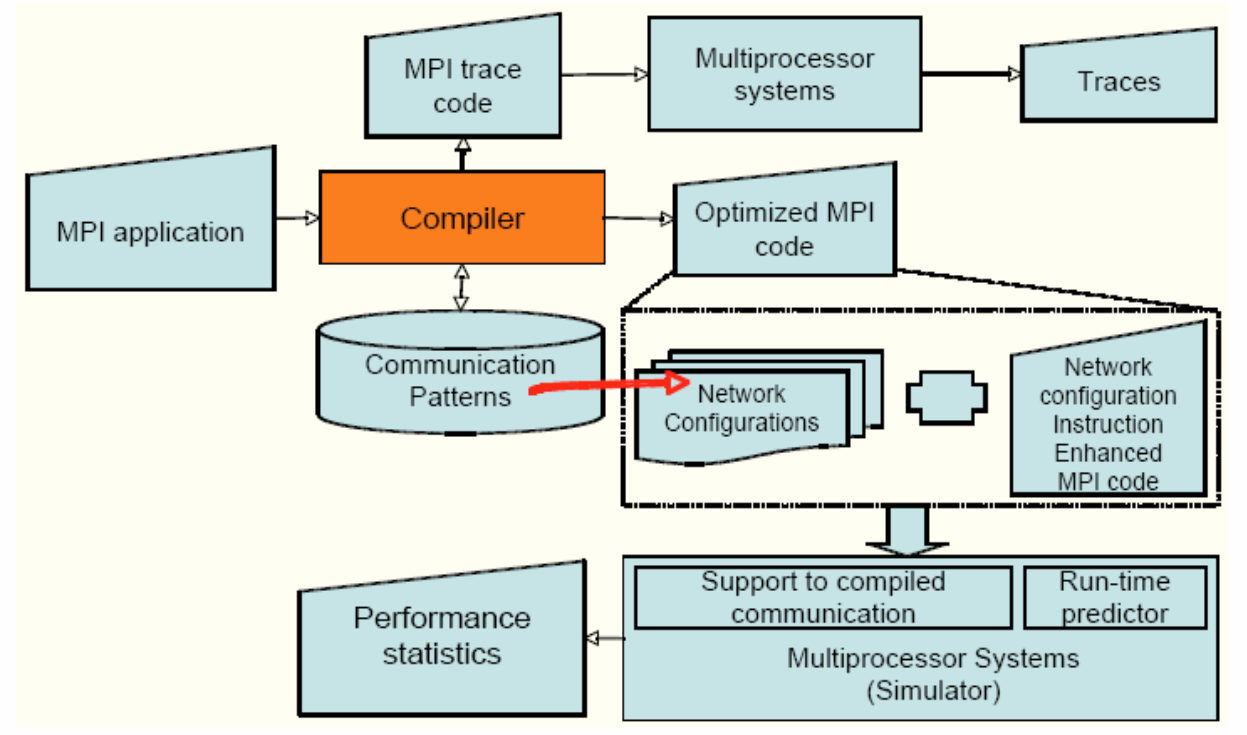


Figure 30. Summary of Performance Analysis Methodology

A fundamental issue of performance analysis is to pick a suitable performance comparison basis. In this research, our goal is to develop a cost-efficient alternative for next generation packet switching techniques in multiprocessor systems. A well recognized candidate of next generation packet switching techniques is crosspoint-buffered packet switching technique. Crosspoint-buffered Packet switching networks can achieve very high performance [46]. Unfortunately, the cost of this type of switch is very high, and makes it currently impractical. However, it sets up a very challenging target for performance investigation in this research. In this work, we simulate FAT Tree networks that implement crosspoint-buffered

packet switching techniques. They are referred to as Fast Packet (**FP**) switching networks in this section.

In our experiments, we report the results of 4 NAS benchmarks, COMOPS256 and a synthetic benchmark SYN256 for performance study. The communication pattern characteristics of these programs are summarized below.

The communication degrees of MG256, CG256, SP256 and BT256 are 13, 5, 6 and 6, respectively. The communication is typically evenly distributed across each node pairs with the exception MG256 whose dominant communication is evenly distributed among only 6 of the 13 pairs. These 4 programs either have only one communication phase or are dominated by a big communication phase.

SYN256 is a synthetic MPI program and its communication pattern contains 3 phases. Each phase performs 2-D mesh communications along 2 dimensions embedded within the same 3-D mesh. Each phase has a communication degree of 4 and the overall communication degree of the entire program is 6. This program sets up message size randomly.

For the ASCI COMOPS benchmark [53], here refer to as COMOPS256 to indicate that we are interested in running it on 256 processors, the point-to-point communication consists of three phases: ping-pong, 2-D ghost cell update, and 3-D ghost cell update. Here the topology of the 2-D communication is not embedded in the topology of the 3-D communication.

These benchmarks are all dominated by static and persistent communication. Table 2 and Table 3 show that all the 4 NAS benchmarks contain either completely static or persistent communication. COMOPS256 and SYN256 have only static communication. All of them belong to the kind of parallel applications that this research targets to.

We have presented the simulated multiprocessor architecture in chapter 8.2 and identified the important system parameters in section 8.2.4. The values used in our experiments are listed in Table 5 except circuit replacement threshold, which will be determined in section 9.2.1.

Table 5. Simulation System Parameters and Values

Parameter	***Value***
Number of processors	256
Processor operating frequency	10GHz
Overhead of MPI_Send and MPI_Recv	5000 cycles
Parameters for packet switching networks	
Link bandwidth	4Gb/s
Per-switch arbitration time	80 ns
Minimal propagation time using only low level FAT tree stage	130 ns
Minimal propagation time using both FAT tree stage	420 ns
Parameters for circuit switching networks	
Link bandwidth	4Gb/s
Propagation delay	200 ns
Connection establishment delay	3 ms

Note that the packet switching network is constructed as a two-level FAT tree. We assume that the first and second level switches are connected with very long wires. This introduces relatively long inter-stage propagation delay. But it is a reasonable assumption given the observation that many supercomputers are hierarchically organized. For example, the lower

level of a FAT tree might exist within a single rack or partial rack of closely connected blade systems while the higher level switches have connections between the racks that are typically much longer [2, 48].

We simulated several different types of multiprocessor systems to study the effectiveness of our compiled communication techniques. The simulation results presented in the remainder of this chapter show that we can achieve the same level, or even higher level of performance than the fast packet switching networks with much less expensive circuit switches using the compiler techniques developed in this research.

9.2.1 Pure Runtime Scheduling

First, we need to have a brief view on the performance that the targeted multiprocessor systems can achieve with pure runtime scheduling. Only when the performance of pure runtime scheduling is well tuned, we can claim the effectiveness of compiled communication techniques.

Through the experiments and analysis in this section, we also want to find the circuit switching system parameters that have more impact on performance than others—the number of circuit switching networks, the circuit replacement threshold, or some other system parameters.

As we have described before, each simulated multiprocessor system deploys a relatively slow packet switched network to accommodate dynamic and low volume messages. In the cases that the circuit switching networks are absent or that the circuit switching networks are not well utilized, the communication subsystem performance is roughly the performance of the slow packet switching network.

Given that the most significant disadvantage of circuits switching is the long establishment delay, how well we can amortize this big latency has a significant impact on the

performance. Thus the circuit replacement threshold, which is the minimal lifetime of a circuit before it can be replaced, is expected to be a very important system parameter for pure runtime scheduling.

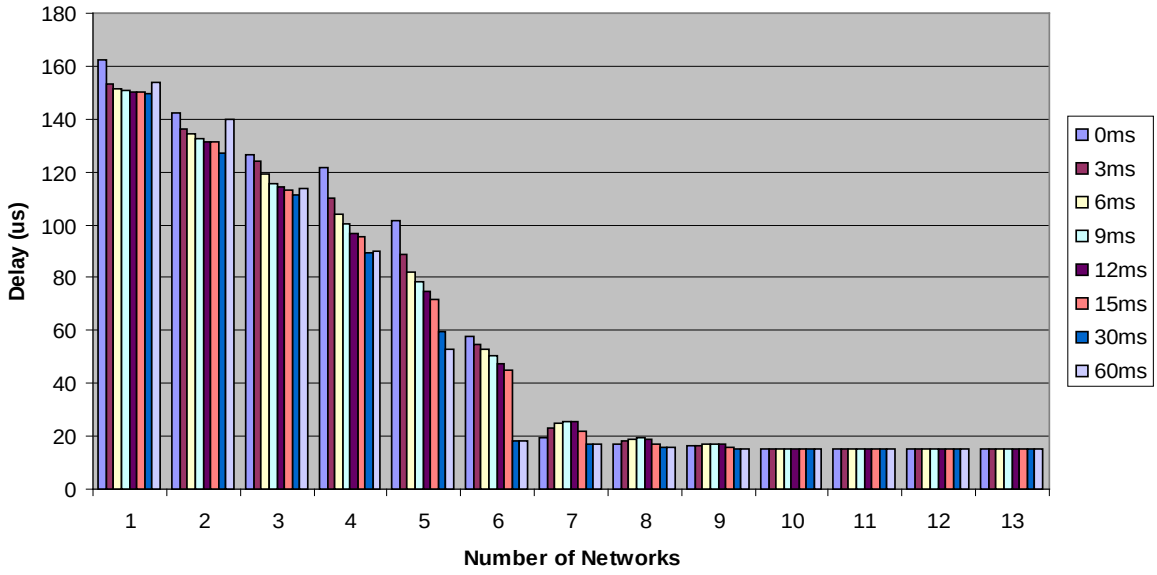


Figure 31. Message Delay of MG256 with Different Circuit Replacement Threshold

In Figure 31, we present the average message delay of MG256 over different circuit replacement threshold and different number of circuit networks. From the result, we can see that picking a reasonable circuit replacement threshold is important to the system performance. Note that the overall communication degree of MG256 is 13. But the communication degree is reduced to 6 by applying the FilterUntilFit operation, whose algorithm is presented in Figure 18, to filter out connections that cause less than 0.1% of the total communications.

From the results, we can conclude that in general the message delay goes down as the threshold goes up when the threshold is not too large. The reason is that increasing the threshold allows a connection to be used for more communications and can effectively relieve circuit

thrashing. However, when the threshold is too large, low-volume circuits can prevent high-volume circuits from being established, and eventually reduce system performance. In Figure 31, this can be clearly seen from the results with 60ms replacement threshold and with 1, 2 and 3 circuit networks.

When the number of circuit networks goes to 4, 5, and 6, the majority of communication can be satisfied by the circuit switching networks. Since most of the high-volume connections are established, larger replacement threshold always improves performance.

When we have more than 6 circuit networks, the message delay continues the down trend but very slowly, as the number of circuit switching networks increases. In these configurations, the circuit network capacity 6 is already enough to serve all the high-volume communications. However, when the runtime scheduler decides to establish a low-volume circuit, the overall communication performance is reduced. Such scheduling decisions are likely to appear because of the runtime execution pace and the scheduled sequence to perform communications. Such runtime noises may overwhelm the impact of the replacement threshold on communication performance in certain system configurations. This is the reason we no longer observe the expected relationship between message delay and the replacement threshold with more than 6 circuit switching networks in Figure 31.

From the results, we also observed that the communication delay in general goes down as the number of circuit networks increase. But when the majority of communications are satisfied by the circuit switching networks, increasing number of circuit networks has no effect to system performance for systems, refers to the cases with 7 to 13 circuit switching networks in Figure 31.

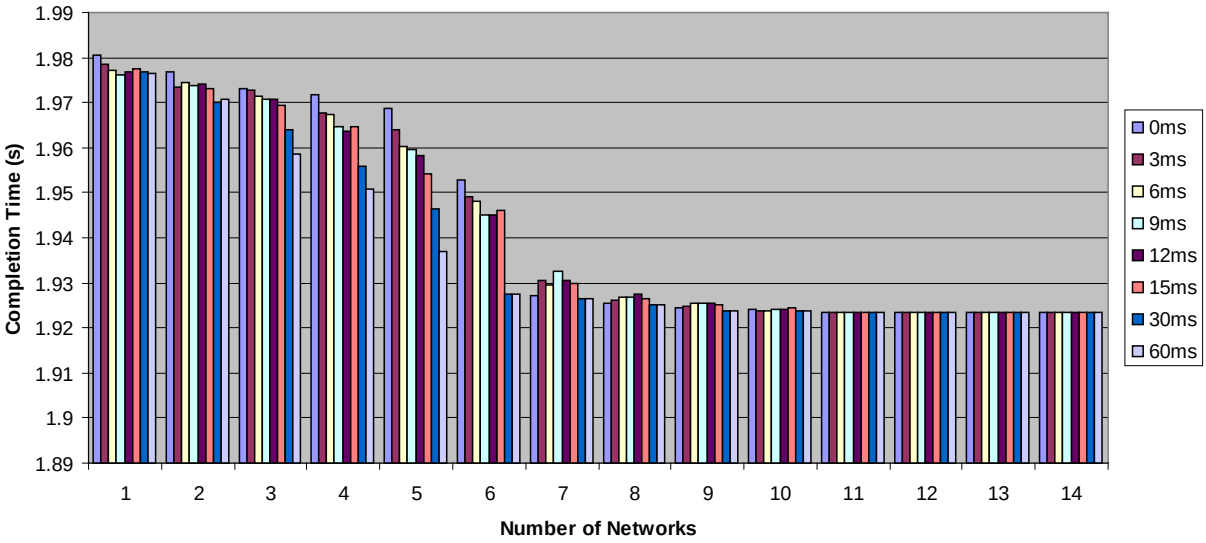


Figure 32. Completion Time of MG256 with Different Circuit Replacement Threshold

The completion times of MG256 with different configurations are show in Figure 32. The results demonstrate generally the same trend as the message delays shown in Figure 31. This is reasonable since in general the completion time should be proportional to the communication delay given the same amount of computation time for a parallel application. However, the execution paces of different processors, the runtime scheduled circuit establishment sequences may have a non-trivial impact on the completion time, especially in relatively small benchmark programs. This explains the slight differences between results shown in Figure 31 and Figure 32.

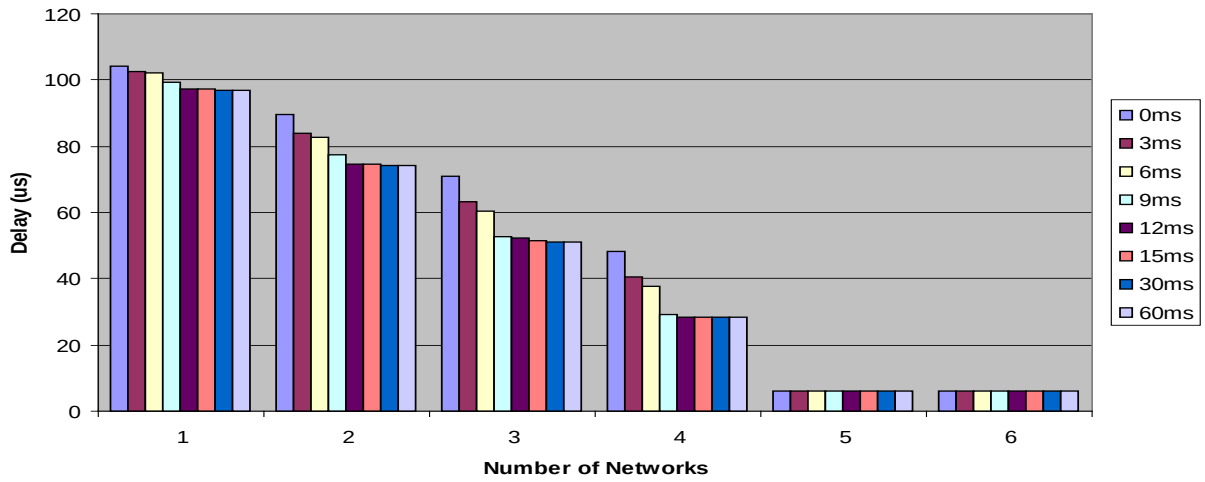


Figure 33. Message Delay of CG256 with Different Circuit Replacement Threshold

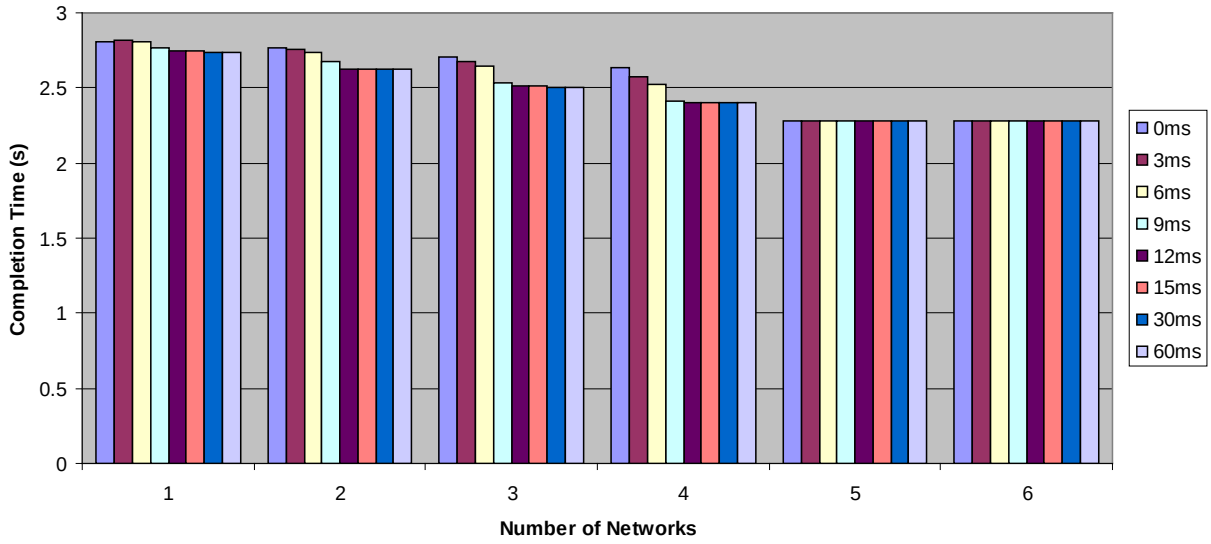


Figure 34. Completion Time of CG256 with Different Circuit Replacement Threshold

The message delay and completion time of CG256 are show in Figure 33 and Figure 34, respectively. As we have described before, the overall communication degree of CG256 is 5 and the communications are evenly distributed among all connections.

The same observations obtained from MG256 experiments are observed again in CG256 experiments. The communication delay goes down as the replacement threshold goes up. Since the communications is evenly distributed across all connections, the performance is always improved as the threshold increases. Also, the performance is always improved as the number of circuit networks increases.

However, the execution times of the benchmark programs are not dominated by communication time. The runtime execution pace and communication operation sequences have a visible impact on the overall completion time. Therefore, the message delay is a much better performance measurement metric for this research. In real large scale parallel applications, communication time is a significant portion of the entire execution time. Thus the message delay results can effectively demonstrate how well the work in this dissertation can improve the performance of real large scale parallel applications. In the remainder of this chapter, we will report the average message delay to demonstrate performance results.

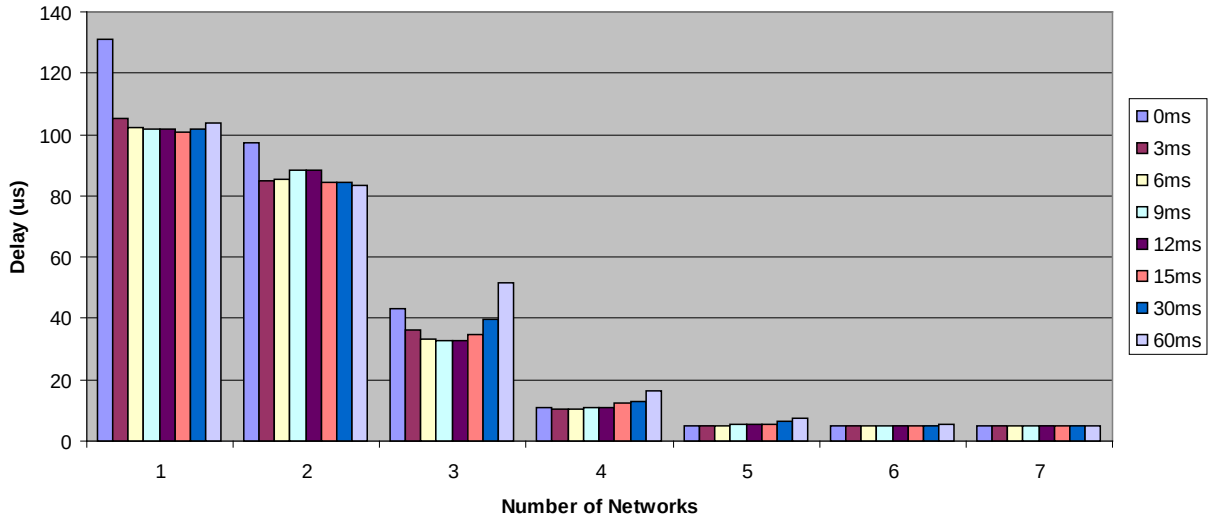


Figure 35. Message Delay of SYN256 with Different Circuit Replacement Threshold

The results of SYN256 are presented in Figure 35. SYN256 has 3 phases, each has a communication degree of 4 and the overall communication degree of the entire program is 6. Note that this program sets up message size randomly. This randomness introduces a small amount of randomization to the results in the configuration with 2 circuit switching networks. Given that the communication degree in each phase is 4, 3 circuit networks theoretically can satisfy only 75% of the entire traffic and we can expect certain amount of collisions and replacements, the replacement threshold has a big impact to performance. We obtained the same conclusion here as from MG256 and CG256 results. Specifically, very small threshold causes serious circuit thrashing and longer average message delay. As the threshold increases the circuit establishment overhead is well amortized by serving more messages, which leads to lower communication latency. When the threshold is too large, the network has less opportunity to serve more communication requests which increases the average message delay. The same phenomenon is observed with other benchmarks.

To summaries, circuit replacement threshold is an important system parameter for communication performance. It can be finely tuned for best performance. But based on observations within this research, 4 times of the circuit establishment latency, ***12 ms***, looks like a generally good choice for most experiments we did. This is an overall estimation and may have to be re-evaluated for other system configurations and applications.

From the above experiments, we also observe that the circuit switching network capacity, simply indicated by the number of circuit networks, is another important system parameter that has significant impact on performance. When the number of circuit networks is larger than the communication degree, almost all the communications are able to be served by the circuit

networks and it may overwhelms the performance impact of some other system parameters, such as the circuit replacement threshold. Most of the parallel benchmark programs, like NAS parallel benchmarks, have very regular communication patterns with pretty small communication degrees.

Large scale parallel applications generally have very regular communication patterns. But they usually have significant amount of very small communications which makes their overall communication pattern degree very large. But when we simply filter out a small percentage, say 10%, of small communications, the communication degree becomes a very small number. The results are reported in [8]. MG256 demonstrates such cases well. Its overall communication degree is 13. By filtering out less than 0.1% of small communications, the effective communication degree is reduced to 6. However, pure runtime scheduling cannot obtain such knowledge and cannot compete with compiler assisted scheduling because of these low-volume-but-high-degree communications.

9.2.2 Different Simulated Systems

We simulate different types of multiprocessor systems to demonstrate the performance of pure runtime scheduling and the benefits of our compilation framework. They are detailed below.

> ***SP*****:** Multiprocessor systems that deploy only the ordinary **s**low **p**acket switching interconnection networks—referred to as SP. Remember that we always include such a slow packet switching network in any multiprocessor system configurations that include circuit switching networks. The purpose of investigate SP is to demonstrate the following facts: (1) SP is really a ordinary packet network, whose performance is far below the next

generation fast packet switching techniques, which is referred to as **FP** and detailed later; (2) the performance we achieved in circuit switching enabled multiprocessor systems comes from techniques developed in this research, not the existence of the slow packet network.

***FP*:** Multiprocessor systems that deploy very **f**ast fully buffered crossbar **p**acket switching interconnection networks are chosen—referred to as FP. They are the targeted multiprocessor systems for performance study in this research. In other words, we compare the performance between what we can achieve with techniques developed in this research and what FP systems can implement to demonstrate that our technique is a promising alternative design for FP in the next generation high performance computing systems. A FP system can include a configurable number of fast packet switching networks. For fairness, each such network is configured as having the same bandwidth as it counterpart circuit switching network. Note that FP systems do not include the ordinary packet switching network which is deployed with circuit switching networks.

***RT*:** Multiprocessor systems that rely on pure **r**un**t**ime scheduler. We have investigated certain aspects of pure runtime scheduling, but we still need to compare it with other multiprocessor system configurations for performance study.

In Chapter 7.0 , we theoretically explored the different runtime approaches to preload compiler identified network configurations. In general, there are two approaches: global network configuration preloading and local network configuration preloading.

***GP*:** Multiprocessor systems that use a synchronized network configuration preloading approach. In such systems, the runtime scheduler synchronizes all processors at the beginning of each communication phase and globally preloads the network configuration before any processor enters that phase. Since the operation of synchronized preloading happens on all nodes at the same time, we refer to this as **g**lobal **p**reloading.

***LP*:** Multiprocessor systems that use an unsynchronized network configuration preloading approach. In such systems, the runtime scheduler does not synchronize all processors at the beginning of a communication phase. The runtime scheduler starts and continues attempting to establish the connections for a processor within a network configuration when the processor is ready to enter the phase and discards all pending requests when the processor exits from the communication phase. Sine the network configuration preloading for different processor occurs theoretically independently, we also call this **l**ocal **p**reloading.

In both GP and LP systems, once a connection is established, it is pinned in the circuit network until the communication phase finishes. This is to prevent runtime scheduling from disturbing the effect of network configuration preloading. Given that most parallel applications in general are synchronized well, the performance of GP and LP is expected to be similar.

PP: In this type of systems, at load-time, the circuit switching scheduler preloads as many connections as possible and pins them in the circuit switching network. The connections are specified in a network configuration for the whole application. Because the connection establishment delay is large in the circuit switching network, the earlier the connections are preloaded, the better the performance that can be achieved. However, the communication degree of a communication pattern in an entire parallel application is usually larger than the network capacity. It may not be possible to establish all the connections required in the configuration.

PPP: In this type of systems, the network configurations are generated for each communication phase. At runtime, the circuit switching scheduler **p**reloads and **p**ins the connections at the beginning of each **p**hase.

Although PPP needs fewer number of circuit switching networks than PP, it does not always achieve better performance than PP. The interval between two adjacent communication phases may not be sufficient for the connections needed by the second phase to be established. Thus, many messages from the second phase may be delivered through the slow packet switching network.

For each system, we run the simulation with different number of circuit switching networks. The simulated system parameters were shown in Table 5. All links in both the fast packet switching networks and the circuit switches have the same bandwidth of 4G bps. We model the circuit switches as optical micro-electro-mechanical system (MEMS) - based switches

whose connection setup overhead is set to 3ms [25, 83, 8]. The slow packet switching network has bandwidth of 400M bps.

9.2.3 Comparison of RT with SP and FP

In this section, we compare the performance of RT with SP and FP. Specifically, we will measure the average message delay as a function of the number of networks. For SP and FP systems, that means the number of packet switching networks in the corresponding configurations. For RT systems, it is the number of circuit switching networks within the multiprocessor systems. Note that there is always an implicit slow packet switching network deployed in all RT multiprocessor systems.

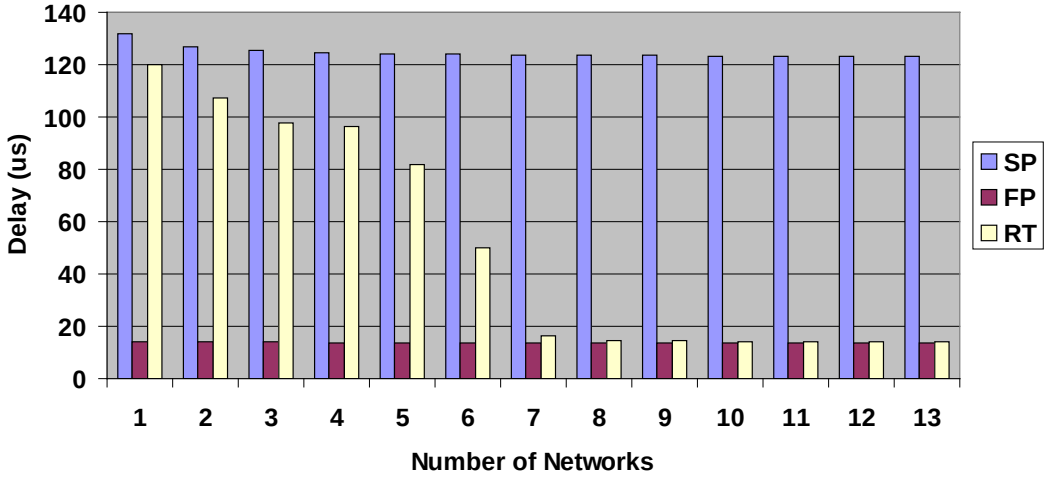


Figure 36. Message Delay of MG256 in SP, FP, and RT

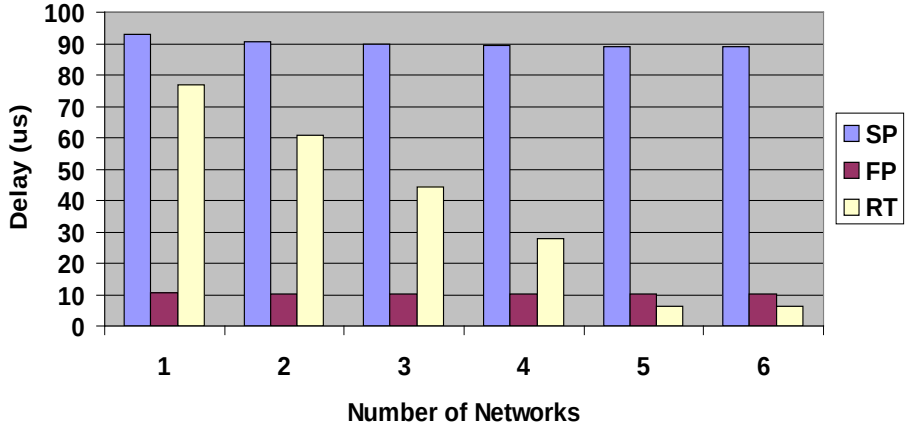


Figure 37. Message Delay of CG256 in SP, FP, and RT

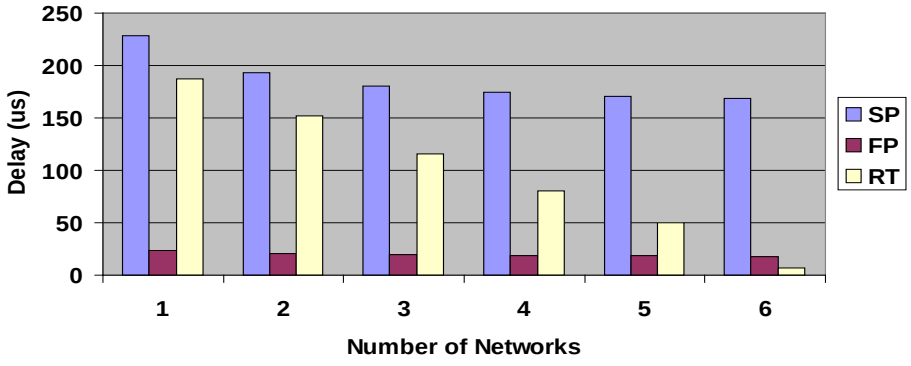


Figure 38. Message Delay of SP256 in SP, FP, and RT

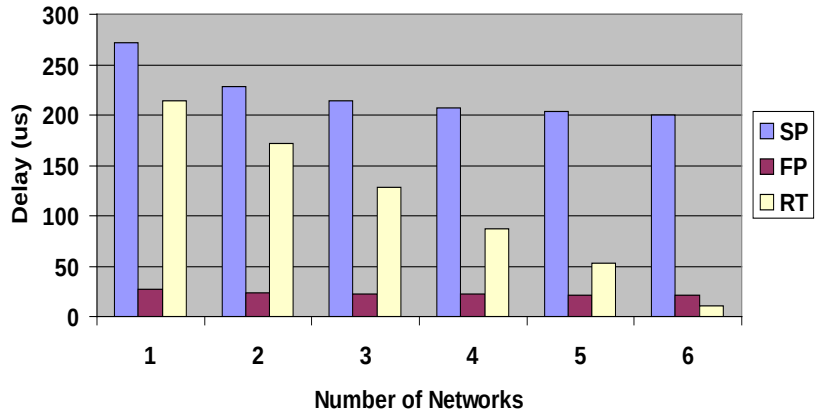


Figure 39. Message Delay of BT256 in SP, FP, and RT

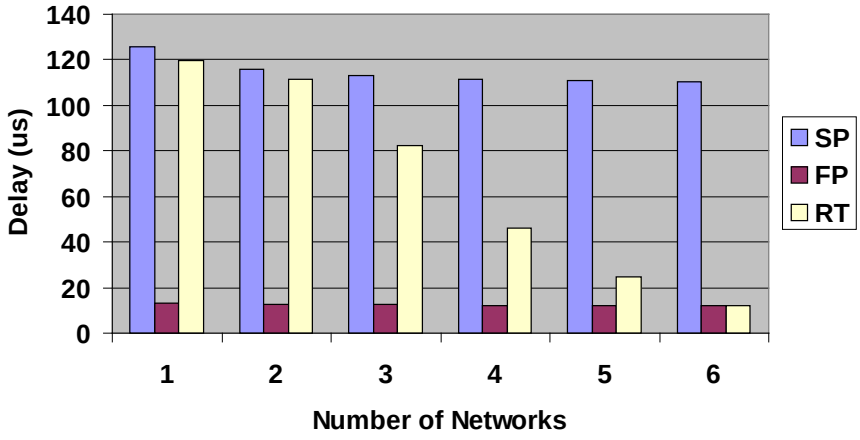


Figure 40. Message Delay of SYN256 in SP, FP, and RT

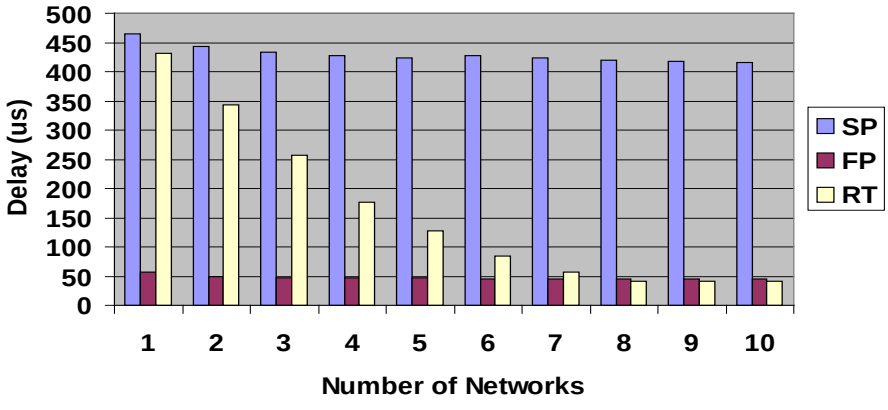


Figure 41. Message Delay of COMOPS256 in SP, FP, and RT

Figure 36 presents the result of the MG256 benchmark. From the figure, we have the following observations.

First, the average message delay measured in SP is much larger than that in FP. This shows that FP has very high communication performance, and a very competitive bar to challenge. It also suggests that the inclusion one SP network in circuit switching enabled multiprocessor systems has no visible impact on communication performance. This is because the communication performance of slow packet networks is so low that it is far below what circuit switching networks can achieve.

Second, the message delay in RT systems goes down as the capacity of circuit switching networks increases. When the circuit switching network capacity is able to serve the majority of communication, which includes the cases with more than 6 circuit switching networks, RT achieves the same level of communication performance as FP.

Third, interestingly, adding additional fast packet switches did not improve the communication delay for two main reasons: (1) the NAS benchmarks produce messages infrequently enough that the majority of messages have left the buffer in the outgoing NIC prior to the arrival of the next message and (2) we assume that individual messages cannot be broken up and sent across separate network planes due to problems like out of order arrival of packets. Given that many parallel application are dominated by static and persistent communication and it is difficult to achieve a significant benefit with multiple packet switch networks, our approach using circuit switching is very promising.

Finally, we need to point out that the performance of the pure runtime scheduling approach, RT, heavily depends on the circuit switching network capacity and the communication distribution. When the communication of a parallel application is unevenly distributed, RT

cannot accomplish competitive communication performance as our compiler techniques. Only when the number of circuit switching networks is equal to or near the overall communication degree, RT can achieve good results, referring to the last column of Figure 36, similar trends are observed in Figure 37 ~ Figure 41. However, most parallel applications have very big overall communication degree but the majority of the communication aggregates at much smaller number of connections. Thus RT may not work well with smaller number of circuit networks.

Figure 37 shows that RT outperforms FP for CG256. This is because the communications of CG256 are evenly distributed across all connections and the messages on the same connection appear pretty frequently. Thus what RT can achieve is almost what the ideal scheduling can achieve.

Figure 38 shows the average message delay for SP256. Note that the message delay visibly decreases as the number of networks increases in SP and FP systems for SP256. This is because that the network traffic generation speed of SP256 is larger than the network bandwidth.

The results of BT256, SYN256, and COMOPS256 are shown in Figure 39, Figure 40, and Figure 41, respectively. Generally, FP sets up a very challenging performance bar to compete with. The communication performance of SP is very low. RT starts with performance near SP and approaches the performance of FP as the number of circuits switching networks increases in a multiprocessor system.

9.2.4 The Effect of Network Loading Approaches

We described in section 7.2.1 two different approaches to load the network configurations for each phase. In this section, we study the difference between synchronized/global and unsynchronized/local network configuration preloading. Specifically, we investigate the

performance of GP and LP and compare that to RT. The measured message delays of MG256, CG256, SP256, BT256, SYN256, and COMOPS256 are shown in Figure 42 ~ Figure 47.

As we have predicted in Chapter 7.0 , parallel applications are usually well synchronized. Thus all the processors enter and exit the same communication phase roughly at the same time. From all the results, we can see that there is no noticeable and consistent performance difference between global network configuration preloading and local network configuration preloading. But in terms of scheduling algorithm complexity and runtime cost, local network configuration is much more expensive than global network configuration preloading. So we can conclude that global network configuration preloading is generally a preferred approach. From then on, we choose global network configuration preloading, GP, when we preload network configurations for each phase.

One interesting result is shown in Figure 45, where the runtime scheduling approach, RT, is the best one. This comes from two reasons: (1) BT256 contains one dominant communication phase plus several very small phases, thus preloading network configuration per phase actually introduces extra delays; (2) the communications of BT256 are evenly distributed among all the used connections, and only a very small amount of runtime predicted and established circuits are torn down because most of them are kept busy since established.

In general, GP and LP achieve much better (MG256, CG256, COMOPS256 and SYN256) or slightly worse (SP256 and BT256) communication performance compared to RT.

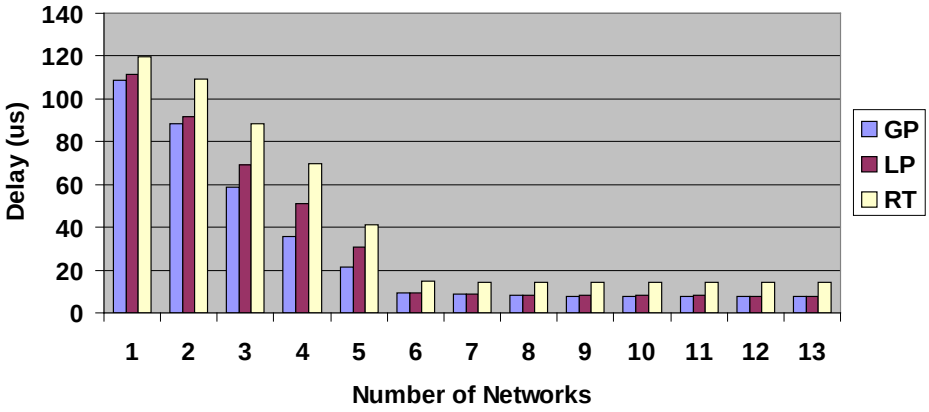


Figure 42. Message Delay of MG256 in GP, LP, RT

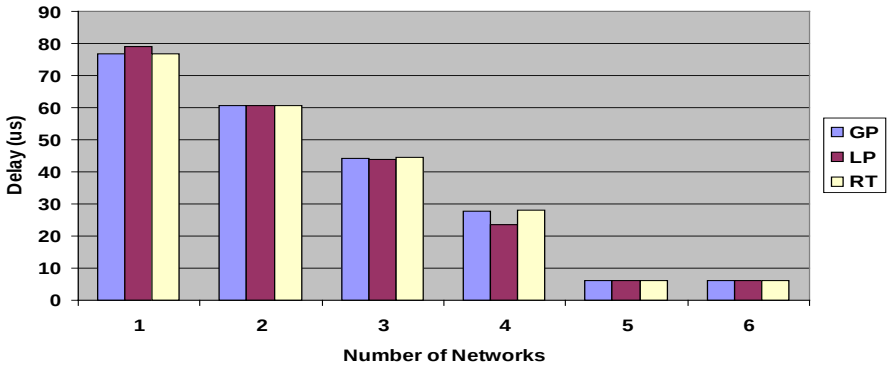


Figure 43. Message Delay of CG256 in GP, LP, RT

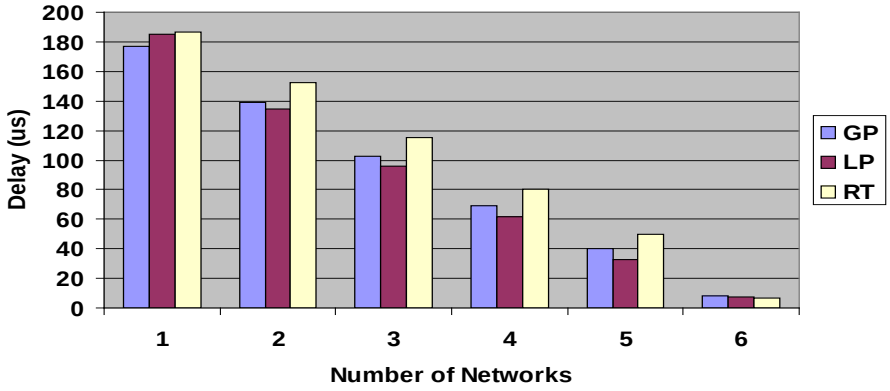


Figure 44. Message Delay of SP256 in GP, LP, RT

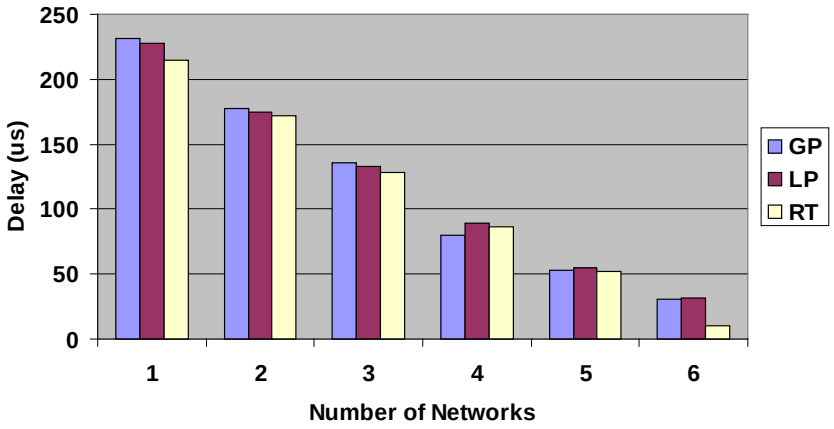


Figure 45. Message Delay of BT256 in GP, LP, RT

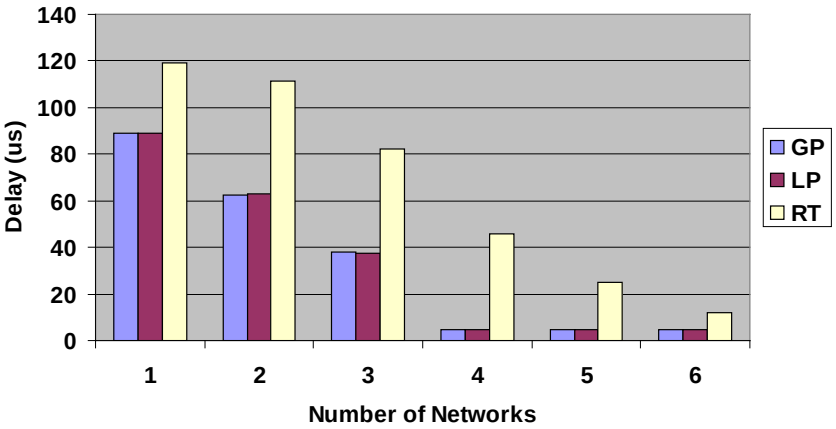


Figure 46. Message Delay of SYN256 in GP, LP, RT

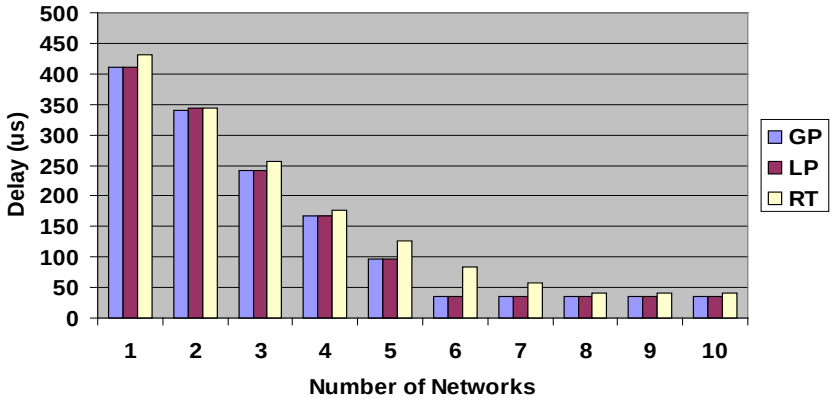


Figure 47. Message Delay of COMOPS256 in GP, LP, RT

9.2.5 The Effect of Phase Partition

In this section, we explore the performance of PP and PPP, and compare them to RT. The experimental results are presented in Figure 48 ~ Figure 53.

It is shown again that as we increase the number of circuit switching networks, we see a near linear reduction in the average message delays for the 4 NAS benchmarks, Figure 48 ~ Figure 51. This trend is maintained for all the applications we have investigated.

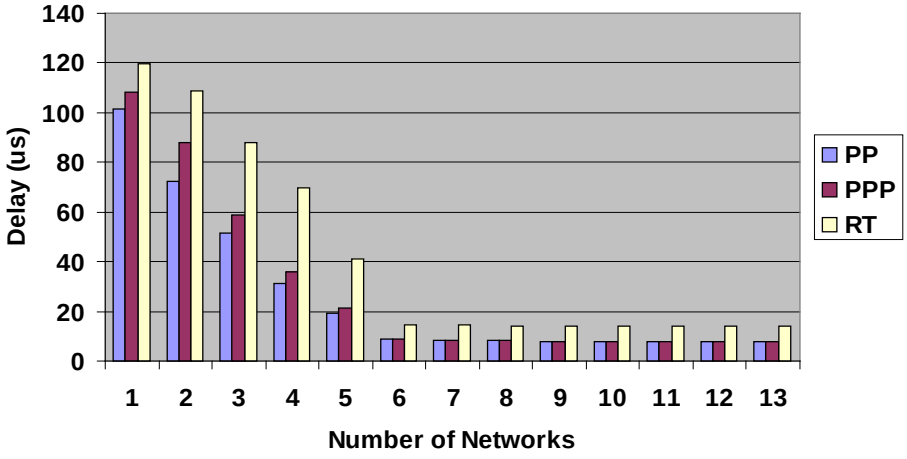


Figure 48. Message Delay of MG256 in PP, PPP, RT

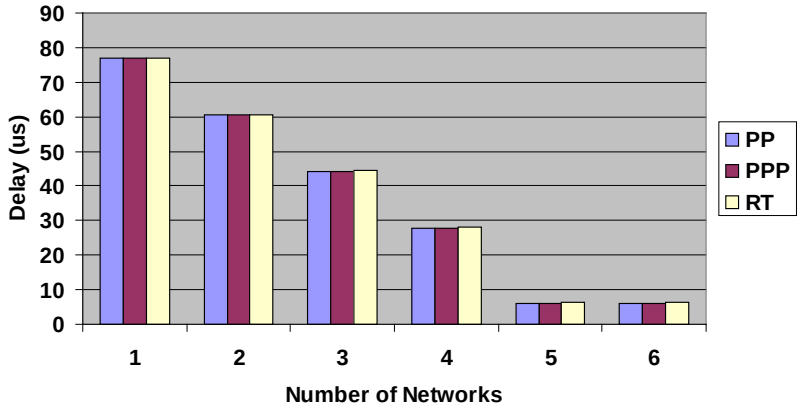


Figure 49. Message Delay of CG256 in PP, PPP, RT

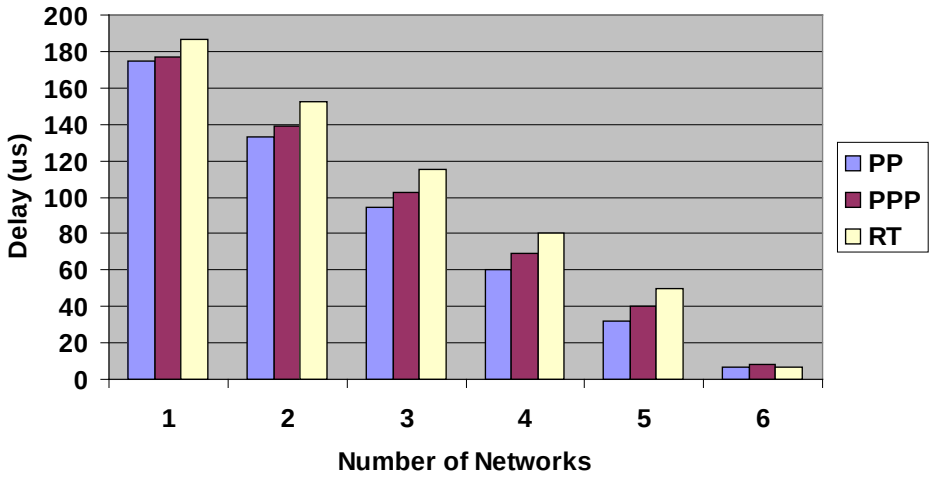


Figure 50. Message Delay of SP256 in PP, PPP, RT

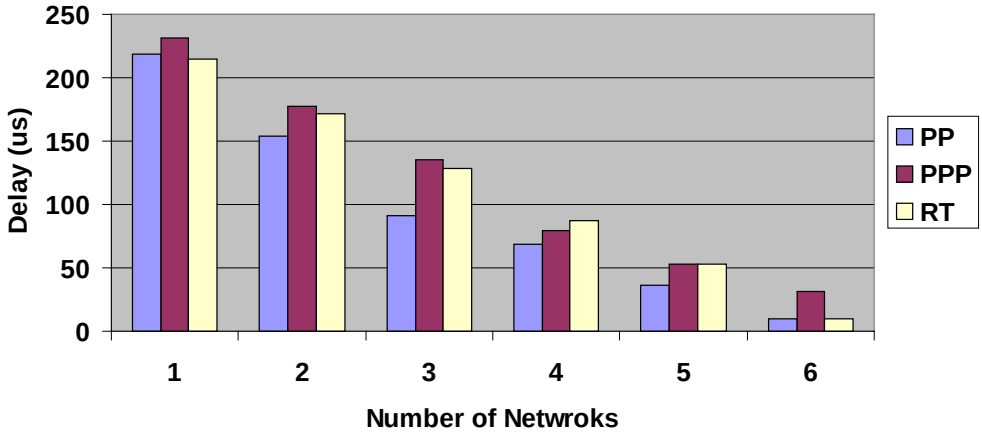


Figure 51. Message Delay of BT256 in PP, PPP, RT

All the applications presented above either have a single phase or have a dominant phase in which communication volume is much larger than other phases. So the phased preloading approach, PPP, performs consistently worse than PP. The reason is straightforward: as the communication pattern of all the 4 applications has only one phase or is dominated by one phase, PPP is either identical to PP or it introduces extra delay by reconfiguring the network at the boundaries of each phase.

PPP can achieve higher performance than PP in certain circumstances. For example, given a communication pattern, in which the i^{th} phase has a communication degree n_i and the global communication pattern has a degree m, it is reasonable to expect that $m > max\{n_i\}$.

It is obvious that PPP can obtain the same performance using only $max\{n_i\}$ circuit switches as what PP can obtain with m circuit switches if the network reconfiguration delay is small or can be overlapped with computation. We use the synthetic program SYN256 and the COMOPS256 benchmark to demonstrate the impact of changing the network configuration between phases.

Simulations show that for both programs the message delay of PPP is lower than PP when using the same number of circuit switching networks and it decreases much faster than PP when we increase the number of circuit switching networks.

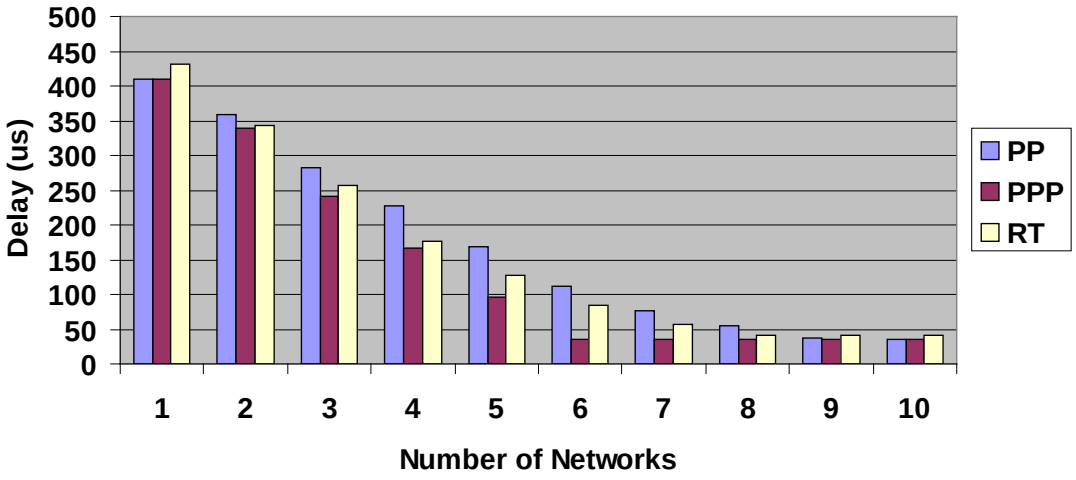


Figure 52. Message Delay of COMOPS256 in PP, PPP, RT

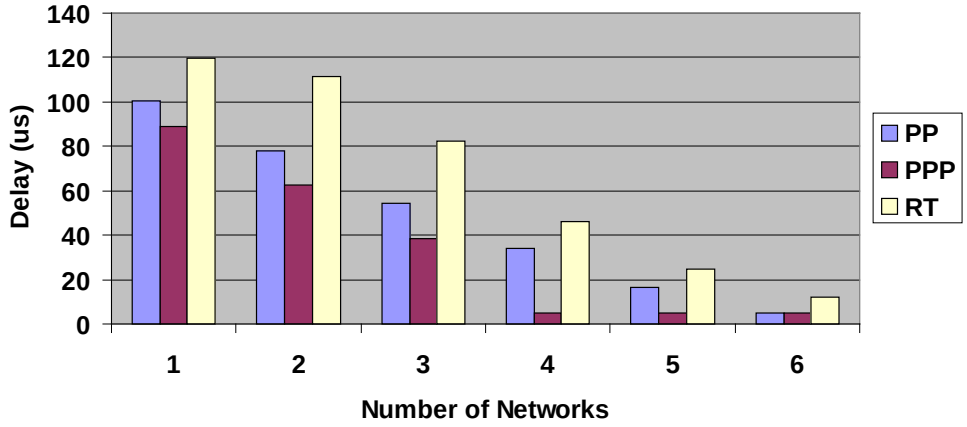


Figure 53. Message Delay of SYN256 in PP, PPP, RT

9.2.6 Comparison of Compiled Communication and FP

Note that the goal of this research is to make circuit switching a cost-efficient alternative to next generation fast packet switching by using the compiled communication approach. Figure 54 presents the performance comparison between compiled communication and FP for all the benchmarks studied before. For each benchmark, we select the lower of PP or PPP average message delay with **6** circuit networks and compare them to FP results.

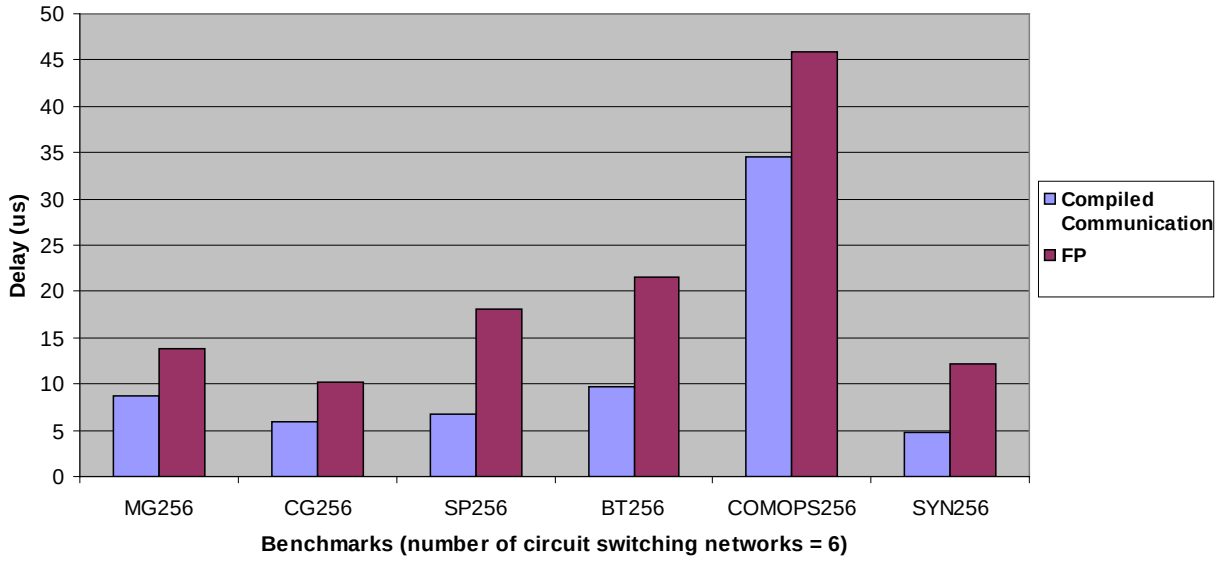


Figure 54. Comparison between Compiled Communication and FP

With compiled communication techniques, we achieved 62.8%, 59.2%, 37.2%, 45.2%, 75.3%, and 39% average message delay of what FP achieved with only 6 circuit switching networks for the 6 benchmarks, respectively. Note that the communication degrees of parallel applications are typically small [47, 8]. Although the maximum communication degree of some parallel applications can be very large, the dominant communication degree may still be small and the low bandwidth connections can be filtered out with minimal performance loss using a circuit switch approach [8]. Therefore, we can expect that 10 or fewer circuit switching networks are sufficient to serve a parallel application in most cases.

9.2.7 Summary

In this chapter, we studied the performance of the techniques developed in this research. It has been clearly shown that our techniques can achieve the same level or much better

communication performance for the next generation high performance packet switching networks. With the assistance from the compiler, circuit switching technique is a promising cost-efficient alternative for the interconnection network design in next generation high performance computing systems.

10.0 CONCLUSIONS

10.1 MERITS OF THIS RESEARCH

Technical advances have brought circuit switching back to a prominent position on the stage of interconnection network design for high performance computing. For example, optical networking is a promising alternative to electronic circuit switching that provides several advantages such as capabilities to handle long wire lengths and achieve high bandwidths.

However, the circuit establishment time is long and the dedication property of circuits prevents other communications from sharing network resources during their usages. Hence, to implement circuit switching with relatively long switching latency mandates a technique to amortize connection establishment overhead. This motivates us to study compiler techniques for achieving efficient low-latency high-bandwidth communications with much cheaper circuit switching technology instead of their expensive packet switching counterparts. Circuit switching becomes very attractive when compiled communication can be utilized for parallel applications. This research serves as a contribution to implement efficient communication on multiprocessor systems with relatively low cost in the high performance computing domain. The major contribution of this research includes:

- This research extends the classification of static and dynamic communications to include persistent communications. Persistent communications are a subclass of

dynamic communications that remain unchanged for large segments of the application execution. It has been shown that typically static and persistent communications are dominant in a parallel application. This suggests that the majority of communication, either static or persistent, can be tackled by compiler techniques.

- A compilation framework for identifying communication patterns of parallel applications from their source code. This framework can be extended to other SPMD parallel programming models in addition to MPI. While developing this framework, compiler challenges for analyzing communication patterns from applications' source code are identified and tackled. Many traditional and new analysis techniques are customized and integrated to fulfill the particular requirements of analyzing SPMD programs, such as traditional CDFG analysis, inter-procedural analysis, determination of static communications, and so on. In this framework, an algorithm for efficiently partitioning communication phases is introduced. Loops are used as the starting point for partitioning phases from which communication patterns with reasonable phase sizes are constructed. Then, communication patterns are effectively mapped into circuit network configurations in terms of connections. Our framework targets static and persistent communications and leaves dynamic communications to an ordinary packet switching network and the run-time connection scheduler.
- Developing a powerful and flexible communication pattern representation scheme. This scheme captures the temporal locality besides spatial locality of communications. It explicitly takes phases into consideration while representing the communication patterns of a parallel application. It can represent both point-to-point communications

and collective communications and is suitable for manipulating communications phases.

- Investigating the mechanism of combining compiler analysis and circuit switching networks. The compiler statically augments the applications with network configuration directives and configuration description files. It is assumed that the multiprocessor system supports compiled communication. Specifically, the scheduler of a circuit switching network interprets the network configuration directives and performs the corresponding network configuration operations at run-time. Also, the scheduler should deploy run-time predictors to control circuit establishment for dynamic communications. A testbed is developed to demonstrate the effectiveness of this research.
- Demonstrating the effectiveness of the proposed research through simulation-based performance analysis. The results show that combination of compiler techniques and circuit switching can provide a promising alternative to packet/wormhole switching for interconnections in high performance computing domain.

10.2 LIMITATIONS OF THIS RESEARCH

This dissertation has several limitations, including limitations of targeted programming model and limitations in the symbolic expression analysis.

- We target explicit message passing, in particular MPI, parallel applications. Although our techniques can be used for other explicit message passing SPMD parallel applications, we cannot handle the implicit message passing parallel applications

directly, such as *shared memory programs*. In shared memory programs, communication patterns can be obtained by examining the memory references. Although it is possible to enhance our communication detection component to make it able to collect all the information involved in the communication operations, it is out of scope of this research.

- Our symbolic expression analysis technique uses static loop analysis to generate symbolic expressions in the loops. Thus, we handle counted loops, i.e., FOR loops in C. For a counted loop, we treat it as a super block in CDFG. If the message destination is defined in a loop other than counted loop, we cannot generate symbolic expressions for it. Such communication operations are classified as dynamic instead of persistent, which may impact the accuracy of our communication pattern analysis. A potential solution to address this problem is to explore the profiling approach. However, our experimental results show that our techniques are accurate enough to achieve similar levels of performance as fast packet switching on circuit switching enabled multiprocessor systems.

10.3 DIRECTIONS FOR FUTURE WORK

There are several open research problems related to this research. In the future, we can extend the work in the following directions:

- Extend techniques developed in this research to other parallel programming models. Although MPI is a very popular parallel programming model in these days, there exist many other parallel programming models which are being used and studied. For

example, OpenMP is still a vital parallel programming model which needs the compiler to automatically create threads for the parallelizable section in a parallel application. Techniques developed in this research may allow the compiler to be able to increase the effectiveness of automatically creating OpenMP threads. UPC has attracted the interests of many researchers for its high productivity and efficiency. Actually, similar methodology can be included in UPC compilers to implement efficient low-cost communication on multiprocessor systems. In general, it is not difficult but very valuable to extend the approach developed in this research to other parallel programming models, especially for SPMD parallel applications.

- Apply this philosophy to compilers for multicore architecture. Currently, multicore architecture has become the standard processor design. In a muticore machine, the data movement latency between different cores and memory modules may be very different. When a machine has multiple sockets, the scenarios become worse: The cores in one socket usually have significantly different data access latency for different modules. The data exchange latency may be unified too. This suggests very similar issues as what we studied in this dissertation: the latency for one pair of CPU cores to exchange data is very different than that of another pair, the latency for a core to access data on one memory module is very different than data on another module. These problems prevent current compilers from fully exposing the hardware-enabled parallelism and potential performance improvement.
- Explore other interconnect subsystems which connect heterogamous resources in supercomputers. This research focuses on the inter-processor interconnects. But given that parallel applications often take huge amount of input data from the storage system

and output huge amount of processed data. When we model storage systems as another kind of system resource for interconnection, there are many different challenges to conquer.

- The power consumption of today's supercomputers is very high. Many powers are consumed by the interconnection networks. Given that buffering can be eliminated from circuit switching devices, it is a reasonable expectation that circuit switching networks can save power. This is also a very interesting topic for the future.

BIBLIOGRAPHY

[1] Top500 supercomputer lists. Top500 Supercomputer sites. http://top500.org/lists.

[2] N. R. Adiga et al. An overview of the bluegene/l supercomputer. In Proc. of Supercomputing (SC), 2002.

[3] A. Afsahi. Design and evaluation of communication latency hiding/reduction techniques for message-passing environments. PhD thesis, University of Victoria, Canada, 2000.

[4] A. Afsahi and N. J. Dimopoulos. Efficient communication using message prediction for clusters of multiprocessors. Concurrency and Computation: Practice and Experience, 12(1):41–50, 2002.

[5] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, , and M. Snir. Sp2 system architecture. In IBM system journal, volume 34, pages 152–184, August 1995.

[6] R. Arlauskas. ipsc/2 system: a second generation hypercube. In Proc. of the 3rd conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues, volume 1, pages 32–48, 1988.

[7] D. Bailey, T. Harris, W. Sahpir, and R. van der Wijingaart. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, December 1995.

[8] K. J. Barker, A. Benner, R. Hoare, A. Hoisie, A. K. Jones, D. J. Kerbyson, D. Li, R. Melhem, R. Rajamony, E. Schenfeld, S. Shao, C. Stunkel, and P. A. Walker. On the feasibility of optical circuit switching for high performance computing systems. In Proc. of SC, 2005.

[9] C. Benveniste and P. Heidelberger. Parallel simulation of the IBM SP2 interconnection network. In Proc. of Winter Simulation Conference, pages 584–589. IEEE, 1995.

[10] J. Brandenburg. Technology advances in the intel paragon system. In Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures, August 1993.

[11] G. Broomell and J. R. Heath. Classification categories and historical development of circuit switching topologies. ACM Computing Surveys (CSUR), 15(2):95–133, 1983.

[12] F. Cappello and D. Etiemble. Mpi versus mpi+openmp on the IBM SP for the NAS benchmarks. In Proceedings of Supercomputing, November 2000.

[13] F. Cappello and C. Germain. Toward high communication performance through compiled communications on a circuit switched interconnection network. In Proc. of the Int. Symp. on High Performance Computer Architecture (HPCA), pages 44–53, 1995.

[14] J.-M. Chang and M. Pedram. Module assignment for low power. In European Design Automation Conf., 1996.

[15] G. Chen, F. Li, M. Kandemir, and M. J. Irwin. Reducing noc energy consumption through compiler-directed channel voltage scaling. In Proceedings of the ACM SIGPLAN 2006 conference on Programming language design and implementation PLDI'06, June 2006.

[16] D. Chiarulli, S. Levitan, R. Melhem, J. Taza, and G. Gravenstreter. Partitioned optical passive star (pops) multiprocessor interconnection networks with distributed control. IEEE Journal of Lightwave Technology, 14(7):1601–1612, 1996.

[17] J. T. Chu. Some methods for simplifying switching circuits using don't care conditions. Journal of the ACM (JACM), 8(4), 1961.

[18] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. ACM Computer Architecture News, 21(2):2–13, May 1993.

[19] V. Delaluz, M. Kandemir, N. Vijakrishnan, A. Sivasubramaniam, and M. J. Irwin. Dram energy management using software and hardware directed power mode control. In IEEE International Symposium on High-Performance Computer Architecture, pages 159–169, 2001.

[20] H. G. Dietz and T. Mattox. Compiler techniques for flat neighborhood networks. In Proc. of 13th Int. Wrokshop on Languages and Compilers for Parallel Computing, 2000.

[21] Z. Ding, R. Hoare, A. Jones, D. Li, S. Shao, S. Tung, J. Zheng, and R. Melhem. Switch design to enable predictive multiplexed switching in multiprocessor networks. In Proc. of the Int. Parallel and Distributed Processing Symp. (IPDPS), 2005.

[22] J. Dongarra. Top10 and awarding. SC06 - Technical Program - BOFs: TOP500 Session. http://top500.org/static/lists/2006/11/bof/TOP500 Nov2006 Jack.pdf.

[23] P. Dowd et al. Lightning network and system architecture. Journal of Lightwave Technology, 14:1371–1387, 1996.

[24] J. Duato, S. Yalamanchili, and L. Ni. Interconnection Networks: An Engineering Approach. Margan Kaufmann, 2003.

[25] P. C. et al. Design and nonlinear servo control of mems mirrors and their performance in a large port-count optical switch. Journal of Micro electro mechanical Systems, 14(2):261–273, April 2005.

[26] A. H. S. C. F. Petrini, W. Feng and E. Frachtenberg. The quadrics network: High-performance clustering technology. 22(1):4657, 2002.

[27] A. Faraj and X. Yuan. Communication characteristics in the NAS parallel benchmarks. In Proc. of the Parallel and Distributed Computing and Systems Conf. (PDCS), 2002.

[28] A. Feldmann, T. M. Stricker, and T. E. Warfel. Supporting sets of arbitrary connections on iwarp through communication context switches. In Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures. ACM Press, August 1993.

[29] M. D. Grammatikakis, D. F. Hsu, and M. Kraetzl. Parallel system interconnections and communications. Boca Raton, Fla. : CRC Press, c2001, 2001.

[30] G. Gravenstreter and R. Melhem. Realizing common communication patterns in partitioned optical passive stars (pops) networks. IEEE Transactions on Computers, 47(9):998–1013, 1998.

[31] T. Gross. Communication in iwarp systems. In Proc. of SC89, pages 436–445. ACM/IEEE, 1989.

[32] C. group. G. t. parallel/distributed ns. http://www.cc.gatech.edu/computing/compass/pdns.

[33] V. Gupta and E. Schenfeld. Combining message switching with circuit switching in the interconnection cached multiprocessor network. In Proc. IEEE Int. Symposium on Parallel Architectures, Algorithms and Networks, 1994.

[34] V. Gupta and E. Schenfeld. Task graph partitioning and mapping in a reconfigurable parallel architecture. Parallel Processing Letters, 5(4):563–574, 1995.

[35] S. Habata, K. Umezawa, M. Yokokawa, and S. Kitawaki. Hardware system of the earth simulator. Parallel Computing, 30(12):1287–1313, 2004.

[36] M. R. Haghighat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. ACM Transactions on Programming Languages and Systems, 8(4):477–518, July 1996.

[37] J. Helin and R. Berrendorf. Analyzing the performance of message passing mimd hypercubes: a study with the intel ipsc/860. In Proceedings of the 5th international conference on Supercomputing, June 1991.

[38] S.-Y. Ho and N.-W. Lin. Static analysis of communication structures in parallel programs. In Proc. of the Int. Computer Symp.(ICS), pages 215–221, 2002.

[39] K. Hwang and Z. Xu. Scalable parallel computing : technology, architecture, programming. Boston : WCB/McGraw-Hill, 1999.

[40] I. S. Institute. U.o.s.c. network simulator ns-2. http://www.isi.edu/nsnam/ns.

[41] K. L. Johnson. The impact of communication locality on large-scale multiprocessor performance. In Proc. of the 19tht Annual Int. Symposium on Computer Architecture, 1992.

[42] B. B. K. D. K. Berkbigler, G. Booker and N. Moss. Simulating the quadrics interconnection network. In Proc. of High Performance Computing Symposium. IEEE, 2003.

[43] V. Karamcheti and A. A. Chien. A comparison of architectural support for messaging in the tmc cm-5 and the cray t3d. In Proceedings of the 22nd annual international symposium on Computer architecture ISCA '95, 1995.

[44] A. Karwande, X. Yuan, and D. K. Lowenthal. Cc-mpi: a compiled communication capable mpi prototype for ethernet switched clusters. In Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'03), 2003.

[45] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. In Computer Networks, volume 3, pages 267–286, 1979.

[46] J. Kim, W. J. Dally, B. Towles, and A. K. Gupta. Microarchitecture of a high radix router. In Proc. of ISCA, pages 420–431, 2005.

[47] J. Kim and D. J. Lilja. Characterization of communication patterns in message-passing parallel scientific application programs. In G. Goos, J. Hartmanis, and J. Leeuwen, editors, Proc. of the Second Int. Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications, pages 202–216, 1998.

[48] A. K. Kodi and A. Louri. Rapid: Reconfigurable and scalable all-photonic interconnect for distributed shared memory multiprocessors. IEEE/OSA Journal of Lightwave Technology, 22(9):2101–2110, 2004.

[49] A. K. Kodi and A. Louri. Design of a high-speed optical interconnect for scalable shared memory multiprocessors. IEEE Micro, 25(1):41–49, 2005.

[50] A. K. Kodi and A. Louri. A new technique for dynamic bandwidth re-allocation in optically interconnected high-performance computing systems. In IEEE Symposium on High-Performance Interconnects, 2006.

[51] D. Lahaut and C. Germain. Static communcations in parallel scientific programs. In Proc. of PARLE, 1994.

[52] J. Liang, A. Laffely, S. Srinivasan, and R. Tessier. An architecture and compiler for scalable on-chip communication. IEEE Trans. on Very Large Scale Integration Systems (TVLSI), 12(4):711–726, July 2004.

[53] LLNL. The asci comops benchmark code. http://www.llnl.gov/asci benchmarks/asci/limited/comops/asci comops.html.

[54] A. MacNab, G. Vahala, P. Pavlo, L. Vahala, and M. Soe. Lattice boltzmann model for dissipative incompressible mhd. In 28th EPS Conference on Contr. Fusion and Plasma Phys, volume 25A, pages 853–856, June 2001.

[55] C. L. McCreary, M. E. McArdle, and J. D. McCreary. Broadcast communication delay metric for the ipsc/2 and ipsc/860 hypercubes. In Proceedings of the 30th annual Southeast regional conference, April 1992.

[56] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, June 1995.

[57] S. Muchnick. Advanced Compiler Design and Implemenatation. Morgan Kaufmann, 1997.

[58] P. Pavlo, G. Vahala, and L. Vahala. Higher order isotropic velocity grids in lattice methods. Physics Review Letters, 80:3960, 1998.

[59] F. Petrini and M. Vanneschi. Smart: a simulator of massive architectures and topologies. In Proc. of int. conf. on Parallel and Distributed Systems(EuroPDS97). IASTED/ACTA Press, 1997.

[60] A. P. H. M. R. Fujimoto, K. Perumalla and G. Riley. Large-scale network simulation how big? how fast? In Proc. of 11th int. Workshop on Modelling, Analysis and Simulation of Computer and Telecommunication Systems. IEEE, 2003.

[61] C. Research. Cray T3D System Architecture Overview. 1993.

[62] M. F. Sakr, S. P. Levitan, C. L. Giles, B. G. Horne, M. Maggini, and D.M. Chiarulli. Predictive control of opto-electronic reconfigurable interconnection networks using neural networks. In Workshop on Massively Parallel Processing Usign Optical Interconnections (MPPOI), 1995.

[63] C. Salisbury and R. Melhem. A high speed scheduler/controller for unbuffered banyan networks. Computer Communications Journal, 24(9):1158–1169, 2001.

[64] S. L. Scott. Synchronization and communication in the t3e multiprocessor. In Proc. Of ASPLOS-VII, 1996.

[65] C. L. Seitz. The cosmic cube. In Communications of the ACM, volume 28, pages 22–33, January 1985.

[66] J. Shalf and D. Bailey. Power efficiency and the top500. SC06 - Technical Program - BOFs: TOP500 Session. http://top500.org/static/lists/2006/11/bof/Top500PowerNov14.pdf.

[67] J. Shalf, S. Kamil, L. Oliker, and D. Skinner. Analyzing ultra-scale application communication requirements for a reconfigurable hybird interconnect. In Proc. of SC, 2005.

[68] S. Shao, A. K. Jones, and R. Melhem. Compiler Techniques for Efficient Communications in Circuit Switched Networks for Multiprocessor Systems. IEEE Transactions on Parallel and Distributed Systems, vol. 20, no. 3, pp. 331-345, Mar. 2009.

[69] S. Shao, A. K. Jones, and R. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In Proc. of the Int. Parallel and Distributed Processing Symp. (IPDPS), 2006.

[70] D. Shires, L. Pollock, and S. Sprenkle. Program flow graph construction for static analysis of mpi programs. In Proc. of Int. Conf. on Parallel and Distributed Processing Techniques and Applications(PDPTA), June 1999.

[71] E. Strohmaier. 28th top500 list. SC06 - Technical Program - BOFs: TOP500 Session. http://top500.org/static/lists/2006/11/bof/TOP500 Nov2006 Erich.pdf.

[72] C. Stunkel, D. Shea, D. Grice, P. Hochschild, and M. Tsao. The sp1 high-performance switch. In Proceedings of the Scalable High-Performance Computing Conference, pages 150–157, 1994.

[73] C. B. Stunkel, J. Herring, B. Abali, and R. Sivaram. A new switch chip for ibm rs/6000 sp systems. In Proc. of SC, 1999.

[74] X.-M. Tian, S. Nemawarkar, G. R. Gao, and H. Hum. Data locality sensitivity of multithreaded computations on a distributed-memory multiprocessor. In Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research. IBM Press, November 1996.

[75] P. Tu and D. Padua. Gated ssa-based demand-driven symbolic analysis for parallelizing compilers. In Proceedings of 1995 ACM International Conference on Supercomputing, 1995.

[76] D. Tutsch and M. Brenner. Minsimulate: A multistage interconnection network simulator. In Proc. of In 17th European Simulation Multiconference: Foundations for Successful Modeling and Simulation, 2003.

[77] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. Journal of Parallel and Distributed Computing, 63(9):853–865, September 2003.

[78] J. C. Wang and S. Ranka. Scheduling of unstructured communication on the intel ipsc/860. In Proc. of SC, 1994, 1994.

[79] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarsinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. ACM SIGPLAN Notices, 29(12):31–37, December 1994.

[80] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarsinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. s. Lam, and J. L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. In SIGPLAN Notices, 1994.

[81] P. H. Worley. Performance evaluation of the ibm sp and the compaq alphaserver sc. In Proceedings of the 14th international conference on Supercomputing, May 2000.

[82] R. G. X. Yuan and R. Melhem. Compiled communication for all-optical TDM networks. In Proc. of the Int. Parallel Processing Sym. (IPPS99), 1999.

[83] T. Yamamoto, J. Yamaguch, R. Sawada, and Y. Uenishi. Development of a large-scale 3d mems optical switch module. NTT Technical Review, 1(7):37–42, October 2003.

[84] X. Yuan, R. Melhem, and R. Gupta. Compiled communication for all-optical TDM networks. In Proc. of SC, 1996.

[85] X. Yuan, R. Melhem, and R. Gupta. Algorithms for supporting compiled communication. IEEE Transactions on Parallel and Distributed Systems, 14(2):107–117, February 2003.

[86] T. yun Feng, C. lin Wu, and D. P. Agrawal. A microprocessor-controlled asynchronous circuit switching network. In Proceedings of the 6th annual symposium on Computer architecture. ACM Press, April 1979.

[87] J. Vetter. Dynamic statistical profiling of communication activity in distributed applications. In Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems. Marina Del Rey, California. ACM Press, Pages: 240 – 250. 2002.

[88] J. Caubet, J. Gimenez et al., “A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications,” Proc. Workshop on OpenMP Applications and Tools (WOMPAT), 2001.

[89] K.C. Claffy, G.C. Polyzos, and H.-W. Braun, “Application of sampling methodologies to network traffic characterization,” Proc. SIGCOMM: Communications architectures, protocols and applications, 1993, pp. 194-203.

[90] S. Shende, A.D. Malony et al., “Portable profiling and tracing for parallel, scientific applications using C++,” Proc. SIGMETRICS Symp. Parallel and Distributed Tools (SPDT), 1998, pp. 134-45.

[91] J.S. Vetter, “Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficiencies,” Proc. ACM Int'l Conf. Supercomputing (ICS), 2000, pp. 245 - 54.

[92] R.E. Kessler, J.L. Schwarzmeier, “Cray T3D: a new dimension for Cray Research”, , Compcon Spring '93, Digest of Papers, 1993, pp. 176-182.

[93] S.L. Scott, G Thorson, “The Cray T3E network: adaptive routing in a high performance 3D torus”, Hot Interconnects IV, 1996.

[94] D. König, Graphok és alkalmazásuk a determinánsok és a halmazok elméletére [Hungarian], Mathematikai és Természettudományi Értesito 34 (1916) 104-119.

[95] R. Cole and J. Hopcroft, On edge coloring bipartite graphs, SIAM Journal on Computing 11 (1982) 540-546.

[96] H. N. Gabow, Using Euler partitions to edge color bipartite multigraphs, International J. Computer and Information Sciences 5 (1976) 345-355.

[97] H. N. Gabow and O. Kariv, Algorithms for edge coloring bipartite graphs and multigraphs, SIAM Journal on Computing 11 (1982) 117-129.

[98] R. Rizzi, Finding 1-factors in bipartite regular graphs, and edge-coloring bipartite graphs, submitted to SIAM Journal on Discrete Mathematics (1999).

[99] G. Viswanathan and J. Larus, “Compiler-directed shared memory communication for iterative parallel applications”. In Proc. of the 1996 ACM/IEEE Conf. on Supercomputing, 1996.

[100] M. F. Sakr, S. P. Levitan, D. M. Chiarulli, B. G. Horne, and C. L. Giles, “Predicting multiprocessor memory access patterns with learning models,” in Proc. of 14th Int. Conf. on Machine Learning, pp. 305–312, 1997.

[101] S.S. Mukherjee and M.D. Hill, “Using Prediction to Accelerate Coherence Protocols.” 25th ISCA, June-July 1998.

[102] M. Kistler , M. Perrone , F. Petrini, “Cell Multiprocessor Communication Network: Built for Speed”, IEEE Micro, v.26 n.3, p.10-23, May 2006.

[103] Lei Jin and Sangyeun Cho. “SOS: A Software-Oriented Distributed Shared Cache Management Approach for Chip Multiprocessors”. *Proceedings of the Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 361~371, Raleigh, North Carolina, September 2009.

[104] Q. Ali, S. P. Midkiff, V. S. Pai. “Modeling advanced collective communication algorithms on cell-based systems”, Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP), AR, 2010.

[105] M. Z. URFIANTO, T. ISSHIKI, A. ULLAH KHAN, D. LI, H. KUNIEDA. “A Multiprocessor SoC Architecture with Efficient Communication Infrastructure and Advanced Compiler Support for Easy Application Development”, IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, Vol.E91-A No.4, pp.1185-1196, April 2008.

[106] J. Miguel-Alonso, J. Navaridas , F. J. Ridruejo. “Interconnection Network Simulation Using Traces of MPI Applications”, International Journal of Parallel Programming, Springer Netherlands, Vol. 37, No. 2, pp. 153-174, April, 2009.

[107] M. Crovella, R. Bianchini, T. LeBlanc, E. Maratos, R. Wisniewski. “Using Communication-to-computation Ratio in Parallel Program Design and Performance Prediction”, Proceedings of the 4th symptom on parallel and distributed processing, Arlington, TX, December, 1992.

[108] X. Qin, H. Jiang, A. Manzanares, X. Ruan, S. Yin. “Communication-Aware Load Balancing for Parallel Applications on Clusters”. IEEE Trsactions on Computers, Vol 59.No. 1, January, 2010.