INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

The Communication Network Simulator for Concordia Parallel Programming Environment

Jun Qu

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

April 2001

Abstract

The Communication Network Simulator

For Concordia Parallel Programming Environment

Jun Qu

Concordia Parallel Programming Environment (CPPE) is a visual environment that can simulate the execution of parallel programs. CPPE is composed of two parts, Concordia Parallel C Compiler (CPCC), which compiles a parallel program into machine code (vCode), and Concordia Parallel System Simulator (CPSS), which executes the vCode generated by CPCC. The communication network simulator, as an important module in CPSS, simulates the message passing procedure in CPPE.

This thesis focused on the design and implementation of CPSS communication network simulator using object-oriented methodology. The aim of the design is to provide an independent, encapsulated, and easy extensible communication network simulator. In the current release of the network module, four types of routing techniques are implemented. They are Shortest Path, Simulated Packet, Packet Switch and Wormhole Routing.

Acknowledgements

I would like to express my special thanks to my supervisor Prof. Lixin Tao for his guidance on the whole process of my thesis work. I am also grateful for his consistent support, advice, encouragement and patience.

My thanks also go to the CPPE team: Prof. Lixin Tao as the team leader, Hassan Hosseini and Ai Kong who built the CPCC system, Hoang Uyen Trang Nguyen and Thien Bui who contributed a lot on CPSS, and Jing Zhang who developed the visual debugger tools which provide the basis for my thesis. With their work, my thesis becomes possible and my work is more efficient.

I appreciate the professors and administrative staffs in the department of Computer Science, especially Prof. Greg Butler who gave the lecture in Software Design Methodologies, Prof. William J. Atwood who gave the lecture in Computer Networks and Protocols, etc., "Monitor" team, whenever I had problems with the computer system, they supported me immediately and gave me the right guidance, Ms. Halina Monkiewicz, her friendliness and administrative support made my student's life easier and pleasant.

My thanks also go to my supportive family. My husband encouraged me throughout my study. I am grateful for my parents' support and help. I also thank to my daughter for her understanding. She had to spend lots of sunny days at home to let me study.

Table of Contents

LIST OF FIGURES ..... **IX**

LIST OF TABLES ..... **XI**

CHAPTER 1 INTRODUCTION **1**

1.1 CPPE OVERVIEW 3

1.2 SCOPE AND OBJECTIVES 5

1.3 CONTRIBUTIONS OF THIS THESIS 6

1.4 THESIS OUTLINE 7

CHAPTER 2 REVIEW OF THE LITERATURE **8**

2.1 PARALLEL COMPUTER SIMULATOR 9

2.1.1 Direct Execution *9*

2.1.2 Direct Execution with Code Augmentation *11*

2.1.3 Functional Simulation *17*

2.2 CPSS SIMULATION TECHNIQUE 20

2.3 ROUTING TECHNIQUES 21

2.3.1 Packet Switching *22*

2.3.2 Virtual Cut-Through *23*

2.3.3 Circuit Switch *23*

2.3.4 Wormhole Routing *24*

2.4 CPSS COMMUNICATION NETWORK SIMULATOR 26

CHAPTER 3 CPSS SYSTEM ARCHITECTURE **27**

3.1 DESIGN OBJECTIVES 27

3.2 SYSTEM ARCHITECTURE 29

3.2.1 General Structure of CPSS *29*

3.2.2 The Code Execution Module *31*

3.2.3 The Debugging Monitor Module *33*

3.2.4 The User Interface Module *35*

3.3 HIGH LEVEL DESIGN FOR COMMUNICATION NETWORK SIMULATOR 38

3.4 MEASURES TO MEET THE DESIGN OBJECTIVES 41

3.4.1 Accurate Simulation *41*

3.4.2 Performance *41*

3.4.3 Flexibility *42*

3.4.4 Repeatability *45*

3.4.5 Correctness Debugging *45*

3.4.6 Performance Debugging *46*

3.4.7 User Friendliness and Portability *46*

CHAPTER 4 THE DESIGN OF CPSS NETWORK SIMULATOR **47**

4.1 NETWORK OVERVIEW 47

4.1.1 Channel Related Parameters *48*

4.1.2 Message Passing Procedure *49*

4.2 CONSIDERATION ON NETWORK SIMULATOR DESIGN 49

4.2.1 Affecting Factors on Network Latency *50*

4.2.2 Situations When Message Passing Is Invoked *51*

4.2.3 Message Type *52*

4.3 NETWORK SIMULATOR DESIGN OBJECTIVES 53

4.4 DESIGN OF NETWORK SIMULATOR 54

4.4.1 Network Simulator Structure *54*

4.4.2 Network Simulator Interface *56*

4.5 WORKING ENVIRONMENT FOR NETWORK SIMULATOR 59

CHAPTER 5 IMPLEMENTATION OF CPSS NETWORK SIMULATOR **61**

5.1 NETWORK PARAMETERS 62

5.1.1 Network Topology *62*

5.1.2 Message 62

5.1.3 Packet 63

5.1.4 Flit 65

5.1.5 Node 66

5.1.6 Virtual Channel / Lane 66

5.1.7 Link 71

5.2 IMPLEMENTATION FOR SHORTEST PATH 74

5.2.1 Constructor and Destructor 75

5.2.2 INIT_NEW_RUN() 75

5.2.3 ROUTING_ROUND() 75

5.2.4 INIT_CHANGE_PHYS_ARCH() 76

5.2.5 netDeadlockRecovery() 76

5.2.6 networkDeadlock() 76

5.2.7 netDeadlockReport() 77

5.2.8 PRINT_PARMS() 77

5.2.9 PARMS_CHANGE() 77

5.2.10 netSendMsg() 77

5.3 IMPLEMENTATION FOR SIMULATED PACKET 79

5.3.1 Constructor/Destructor 80

5.3.2 INIT_NEW_RUN() 80

5.3.3 ROUTING_ROUND() 81

5.3.4 INIT_CHANGE_PHYS_ARCH() 81

5.3.5 netDeadlockRecovery() 81

5.3.6 networkDeadlock() 81

5.3.7 netDeadlockReport() 82

5.3.8 PRINT_PARMS() 82

5.3.9 PARMS_CHANGE() 82

5.3.10 netSendMsg() 82

5.4 IMPLEMENTATION FOR PACKET SWITCH 83
5.4.1 Constructor/Destructor *84*
5.4.2 INIT_NEW_RUN() *84*
5.4.3 ROUTING_ROUND() *85*
5.4.4 INIT_CHANGE_PHYS_ARCH() *85*
5.4.5 netDeadlockRecovery() *85*
5.4.6 networkDeadlock() *86*
5.4.7 netDeadlockReport() *86*
5.4.8 PRINT_PARMS() *86*
5.4.9 PARMS_CHANGE() *86*
5.4.10 netSendMsg() *87*
5.5 IMPLEMENTATION FOR WORMHOLE ROUTING 87
5.5.1 Constructor/Destructor *88*
5.5.2 INIT_NEW_RUN() *88*
5.5.3 ROUTING_ROUND() *89*
5.5.4 INIT_CHANGE_PHYS_ARCH() *89*
5.5.5 netDeadlockRecovery() *89*
5.5.6 networkDeadlock() *90*
5.5.7 netDeadlockReport() *90*
5.5.8 PRINT_PARMS() *90*
5.5.9 PARMS_CHANGE() *90*
5.5.10 netSendMsg() *91*
CHAPTER 6 EXPERIMENTAL RESULTS **92**
6.1 EXAMPLE 1: RING COMMUNICATION PROGRAM 92
6.2 EXAMPLE 2: MATRIX MULTIPLICATION PROGRAM 94
CHAPTER 7 CONCLUSION AND FUTURE WORK **97**
REFERENCES **99**

List of Figures

Figure 1: Multiprocessror and Multicomputer Architectures 2

Figure 2: Structure of CPPE 5

Figure 3: General Structure of CPSS 30

Figure 4: CEM Structure 31

Figure 5: GUI under Windows system 36

Figure 6: GUI under Unix system 37

Figure 7: Structure of Communication Network Simulator 40

Figure 8. Overview of message passing procedure 49

Figure 9: Class Structure of Communication Network Simulator 55

Figure 10. The Usage of CNetwork Type Pointer 56

Figure 11: Class CNetwork Definition 58

Figure 12. Interaction between Network Simulator and the rest of CPSS 60

Figure 13. Two virtual channels sharing a physical link 67

Figure 14. Virtual channel abstraction (Lane) 68

Figure 15. Link Numbering for Two Dimension Topology 72

Figure 16. Class CShortestPath Definition 74

Figure 17. Class CSimulatedPacket Definition 79

Figure 18. The Structure of BusyList 80

Figure 19. CPacketSwitch Class Definition 84

Figure 20. CWormhole Class Definition 88

Figure 21. Ring Communication Program 93

Figure 22. Matrix Multiplication Program 95

List of Tables

Table 1. Performance Statistics for Ring Communication Program 93

Table 2. Performance Statistics for Matrix Multiplication Program 96

Chapter 1

Introduction

Since the first electric processor was built in the ENIAC computer in 1945, computer science had focused on sequential programming for more than 40 years. Despite the tremendous increasing in computing power, there is a wide range of important computational problems in science and engineering that require much greater computing speed. One approach to meet this need is to put many of the standard processors together in one computer. To have more than one processor operating at the same time in the same computer is the basic concept of parallel computer.

Currently, there are two major architectural approaches to fulfilling this requirement: shared memory and message passing. In shared memory computer, called *multiprocessor* computer, all processors access the same data stored in memory. As the memory can not accommodate all the requests simultaneously, *memory contention* becomes the major difficulty in shared memory multiprocessors. To avoid this problem, a distributed-memory parallel computer, called *multicomputer*, is introduced. With this architecture, all processors have their own local memories, and there is a communication network to link all processors together. The interaction between processors is acquired via message passing through this network.

Since we are studying parallel programming, both architectures will be discussed. As multicomputer programming is more complex not only because the program must do explicit message passing between processors, but also the factor that the processor interconnection topology must be considered, we will focus on the multicomputer programming. The multiprocessor and multicomputer architectures are illustrated in Figure 1.

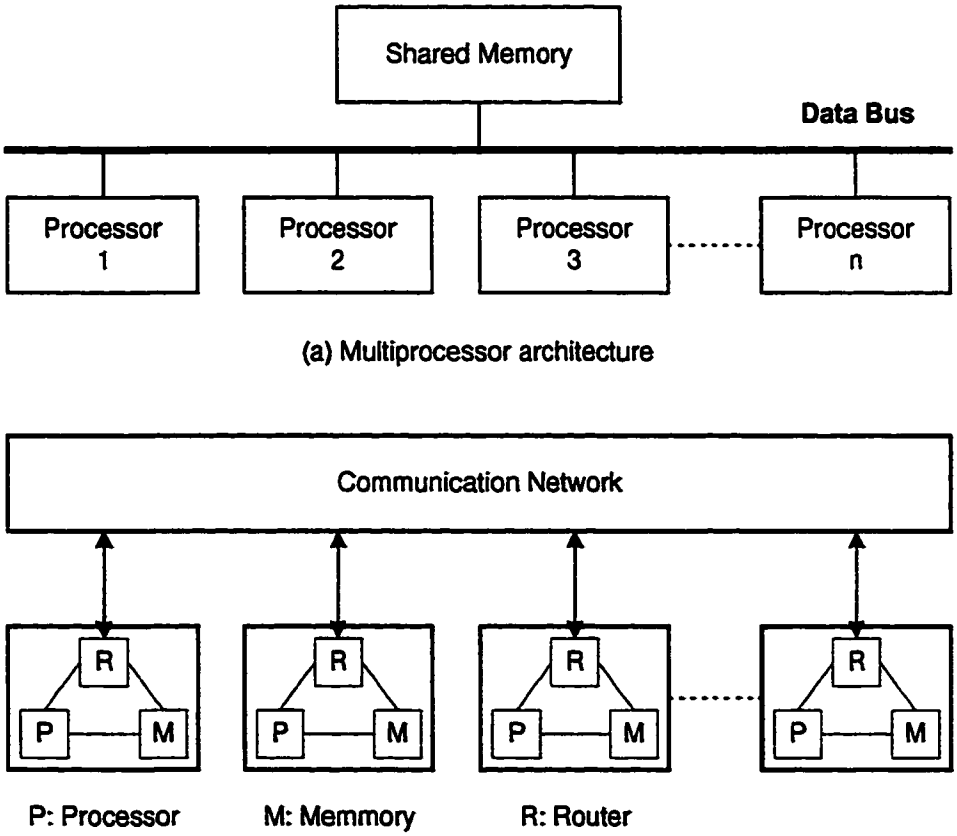

Figure 1: Multiprocessror and Multicomputer Architectures

Building real parallel computer system with multiple configurations is too expensive, but pure mathematical modules are too abstract for complex systems. Simulation is a good compromise. Our parallel programming study is based on a simulation model, which is developed and implemented by Concordia parallel programming research group, named Concordia Parallel Program Environment (CPPE). The purpose to develop this CPPE is to provide users with flexible and efficient software tools to practice on parallel programs, evaluating and optimizing performance of parallel applications.

From previous paragraphs, we have already known that the message passing communication network organization forms the fundamental of multicomputer architecture. In this thesis, we will demonstrate the design of the communication network simulator, and study how the system topology and routing technique affect the performance of parallel programming. Before we continue, we present the structure of our parallel programming environment - the CPPE first.

1.1 CPPE Overview

CPPE is a simulation tool designed to simulate parallel programming in the real world. It contains not only the performance analysis, but also a debugging tool to help users run their parallel program correctly and efficiently. CPPE consists of two main modules:

1. Concordia Parallel C Compiler (CPCC): This module takes the parallel program written in Concordia Parallel C (CPC), which is the extension of standard C code, as input, and then compiles it to an virtual machine code (called vCode), which can be recognized by CPSS.

2. Concordia Parallel System Simulator (CPSS): This module takes the virtual machine code as its input, simulates execution of the application program, yields program output, debugging information, and performance statistic report.

These two modules form the core of CPPE. CPCC performs as a regular compiler. CPSS contains all functionality CPPE provided. It consists of three major sub-modules.

1. Code Execution Module (CEM): The CEM is the processing element of a simulated system. It executes vCode generated by CPCC.
2. Communication Network Simulator: The role of the communication network simulator is to allocate network resources to messages, route and deliver messages, and also detect deadlock in the network if applicable.
3. Utility Module: It contains the Debugger Monitor and the Graphic User Interface (GUI). The debugger monitor provides a set of software tools to facilitate program testing and debugging. The GUI provides friendly user interface to easy parallel programming and debugging.

Figure 2 shows the structure of CPPE and the interactive relationship between each module.

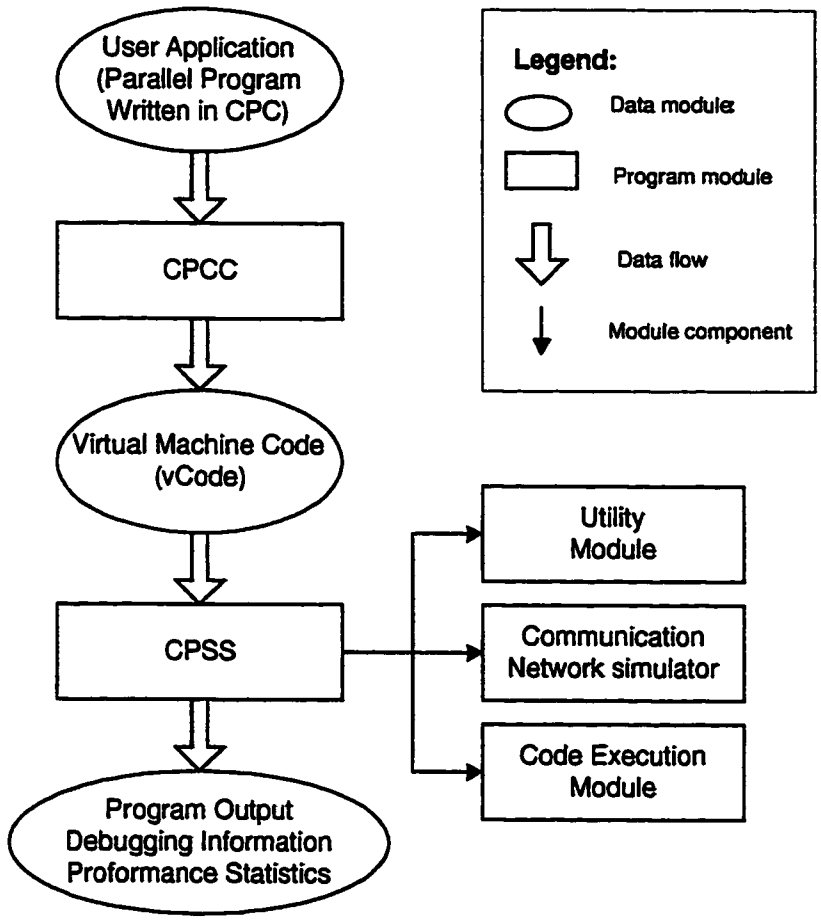

Figure 2: Structure of CPPE

1.2 Scope and Objectives

The scope of this thesis is to design and implement CPSS communication network simulator. The CPSS communication network simulator is designed using objected oriented methodology and implemented using C++ language. Through user interface, user can select different kinds of network routing techniques and system topologies, and

the parallel program can run based on these system configurations. The objective is to provide user a software tool for better understanding how does the network latency influence on the performance of parallel program and how do different network routing techniques gain speedup on parallel program.

1.3 Contributions of This Thesis

In this thesis, four types of routing technique are implemented for the communication network simulator. They are

1. Shortest Path: This is the simplest but least accurate network routing technique. When a message is passed from source node to its destination node, the smallest number of hops is counted between these two nodes. Considering the pre-defined network delay between two adjacent nodes, we get the network latency for this message.
2. Simulated Packet: Simulated packet is similar to shortest path, but the network congestion is considered.
3. Packet Switch: When a message is going to be sent from its source node to its destination node, it is split to packets first. Instead of forwarding the whole message from one node to another, packets are forwarded.
4. Wormhole Routing: It is similar to packet switch, but even more small data unit - flits are forwarded from one node to another. A message, in wormhole routing network, is divided into several packets, and then a packet is further divided into several flits. This routing technique is widely adopted by parallel simulation. It increases the degree of parallelism.

By using the objected oriented software design methodology, a base class, *CNetwork* class, is declared for the network simulator. The diverse versions of network simulator with different routing techniques are derived from this base class. Therefore, CPSS does not need to be aware of which routing technique the network simulator is running on. It just calls the network simulator functions through network simulator interface. The right network simulator functions are linked automatically to CPSS.

1.4 Thesis Outline

Chapter 2 of this thesis provides a literature review of the multicomputer simulations and communication network simulators. A comparison analysis is included. Chapter 3 describes the basic component and key implementation concept of CPSS, which is the base for the implementation of interconnection communication network simulator. Chapter 4 presents the design of the CPSS communication network simulator. The architecture of the CPSS communication network simulator design and the deployed methodology are discussed. Chapter 5 describes the key implementation details of CPSS communication network simulator. Chapter 6 gives the experimental results and summarizes the influence on communication network simulator with different of routing techniques and system topologies. Chapter 7 provides a conclusion of this thesis and lists some suggestions for future work.

Chapter 2

Review of the Literature

From Chapter 1 we already know the structures of multiprocessor and multicomputer and their difference. Nowadays, networks are key components of all computer and communication systems. High performance networks use point-to-point links rather than shared media, in order to exploit parallelism and avoid turn-around and arbitrary delays. These networks are built out of packet switches or routers, interconnected by point-to-point links.

The parallel system simulator developed by our group, the CPSS, is intended to provide accurate simulation of both computation and communication activities. It should provide an accurate, a flexible, and a user-friendly environment for correctness and performance debugging of parallel programs. Thanks to members of Concordia Parallel research group for building the excellent code execution simulation environment and user-friendly interface both under Unix platform and Windows. With their effort, the implementation of communication network simulator becomes possible.

In this chapter, a literature review of parallel computer simulation technique is presented firstly. Then performance affected by network simulation will be studied.

2.1 Parallel Computer Simulator

There are several parallel simulation systems have been developed [1] [9] [10] [12]. According to the simulation techniques, they can be classified into three categories:

1. Direct Execution:
2. Direct Execution with Code Augmentation:
3. Functional Simulation

In this section, all those simulation techniques are examined in details, including their characteristics, advantages, drawbacks, and example simulators.

2.1.1 Direct Execution

With this simulation technique, a parallel program is first compiled into object code, which is in the assembly language of the host computer. During compilation, the compiler identifies two kinds of instructions for the purpose of simulation: local instructions and non-local instructions. An instruction is local if it accesses only local memory and has effect only on its own processor. On the other hand, a non-local instruction is impact another part of the system, such as other processors and networks. The typical non-local instructions in parallel programming include process creation/termination, message sending/receiving, and process synchronization. In the process of simulation, local instructions are executed directly by host processors and timed with the host's machine clocks, while non-local instructions are simulated via a procedure call, which interprets the instruction at the functional level. That is, the

behavior of each non-local instruction is emulated by a host routine as if the instruction were running on the real remote system.

Direct execution technique is generally faster compared with the functional simulation (See 2.1.3), because local instructions are executed directly instead of being interpreted. However, it suffers from a major drawback: difficult debugging and inaccuracy.

The debugging difficulty is caused by local instructions which are executed directly by the host machine, and the simulation engine has no much control on it. It is very difficult to establish connection between user application and the low-level simulation activities. Such connection is important for in-session debugging and fine-tuning of an application. (In-session debugging refers to the debugging interaction between the user and the program during execution of the program. Stepping through the program, setting breakpoint, and tracing variables after breakpoints are examples of in-session debugging).

The low accuracy is caused by two major factors: the way execution time is counted and the monitoring code. As the simulation is timed with the host's clock, which is usually the workstation clock with coarse granularity, the execution time of an instruction is often truncated to the nearest milliseconds (Within a millisecond, the target parallel computer may have executed thousands of instructions or sent hundreds of messages). So, to use the host's clock to estimate simulation execution time is a very raw approximate. For monitoring code, it is often necessary because of the difficulty of in-session debugging. It is used to print out the intermediate result, and it should not be executed on the target

machine. Since there is no way to distinguish monitoring code from the application code, the execution time for monitoring code is also counted to the simulation execution time. This makes the accuracy of the parallel simulation even worse.

In summary, the direct execution technique is usually used for studying hardware feature or network performance evaluation. It is not suitable for accurate simulation of both computation and communication activities on a target multicomputer. In addition, debugging tools provided by direct execution simulators are very limited and based primarily on monitoring code added to the application and simulation engine, it is impossible to provide a user-friendly debugging environment.

An example simulator using direct execution technique is the CARE simulator [9] [10], which simulates the execution of LISP (LISt Processing) code.

2.1.2 Direct Execution with Code Augmentation

This is an enhanced version of direct execution technique. By adding cycle counts (the time that target system would take to execute an instruction) of a local instruction to the object code during compilation phrase. Cycle count of object code will be accumulated during simulation as if the code were being executed on the target parallel computer. The simulation of local instructions is no longer timed with the host's clock but cycle counts. This results in a more accurate simulation than the pure direct execution technique.

The direct execution with code augmentation improves the accuracy simulation problem, but the debugging difficulties are still there. Code augmented technique does not improve the debugging behavior of direct-execution simulator. Monitor code is still used as an approach for in-session debugging. It does not cause any side effect except that the added code may slow down the simulation. However, a simple addition will change the behavior of the simulation since the monitor code is also included in the cycle counting. Conditional compilation flags or macros can be used to execute the cost of the added monitoring code, but it may change the behavior of the application. This is because the additional code may affect the surrounding code indirectly. For example, if addition code uses the registers, the surrounding code may spill more registers than the application without monitor code. This could increase the cost and change the behavior of the system. The more debugging or statistics traces are required, the more perturbed the simulation can be.

Example simulators using direct execution with code augmentation technique are Proteus [2], Tango [8] [13], EPPP [12] [23], and PARSE [21].

Proteus

Proteus, developed at MIT in 1991, uses code augmentation to count the cycles required by the target machine to execute local instructions. The application program is first compiled into the host's assembly language. Then the compiled code is divided into basic blocks of local instructions. A basic block is the smallest block of code delimited by a non-local instruction or an instruction where the execution can branch (e.g. a jump, a

function call). Each instruction in a basic block is matched to the cycle count by looking up a table. The cycle counts of all instructions in that basic block are summed and a code-augmentation program will then add these cycle counts at the end of the block. The cost of each basic block is thus a fixed number and determined at compiler time.

Proteus' debugging capacity depends heavily on the use of sequential *dbx* tools. The user is also allowed to add monitoring code into the simulation engine and the application. During program execution, monitoring code produces data and event traces, and logs the traces into an output file. When the program execution is completed, a graph generator is used to interpret the trace file data and present the results of the simulation.

Although the Proteus simulator is fast, it suffers from several drawbacks.

- Even though the simulation result is improved compare with pure direct execution, the timing results may not be accurate because the cost of each basic block is determined at compile time and is a fixed number. In reality, the cost of an instruction depends on other run-time factors such as the operands (or cache hits if the target machine is shared-memory architecture).
- The simulator can simulator accurately only a limited set of architectures whose instruction sets are similar to that of the host. If the instruction set of the target machine is quite different from that of the host, the assignment if a cycle count to every local assembly instruction is no longer accurate.
- Simulation performance may be substantially degraded if the application involves many processes. In addition to the simulation engine process, a host process is created

for each application process. If the number of application process is large, this may incur substantial overheads of context switching among the simulating threads. In practice, augmentation overhead is an insignificant part of simulation cost. Simulating non-local instructions and context switching dominate the cost of simulation.

- The simulator is not flexible from the user's point of view. When the architecture is changed, the engine parameters must be modified. The engine is then re-compiled and linked with the user application. This is not convenient, especially for experimenting with program mappings. This experiment could require to run the same program on different architectures of varied sizes. The simulator must be re-compiled and linked with the application code each time the topology or system size is changed.
- Debugging capacity relies mainly on software instrumentation. In-session debugging facility is very limited and depends on sequential *dbx* tools.

Tango

Tango simulator was built at Stanford University in 1990. Tango and Proteus were developed independently, but they are quite similar. However, Tango simulates only shared-memory architectures. It was implemented for studying shared memory behavior, shared memory synchronization and concurrency abstractions and for architecture evaluation.

With Tango simulator, application programs are written in C or Fortran. Parallel features are provided by macros. The compilation process consists of five steps: macro expansion, compilation into assembly language, code augmentation, assembly and linkage. If a

parameter needs to be changed, the simulation engine must be modified and the compilation process is repeated.

Like Proteus, Tango may produce inaccurate simulation results due to fixed costs of local instruction blocks calculated at compilation time.

Unlike Proteus, Tango's performance is not as good as Proteus'. Tango uses Unix processes to simulate parallel execution while Proteus uses faster lightweight processes managed by the simulation engine. The context switch time is significant in Tango. In addition, Tango does not support in-session-debugging tool. Debugging and statistics data are provided using the instrumented software approach. Many kinds of trace files are generated. Program outputs are logged in an output file. There are also summary files and event trace files. This is not a user-friendly debugging environment and is not adequate to provide code development or performance fine-tuning.

EPPP Project

The EPPP (Environment for Portable Parallel Programming) simulator is an extended version of Proteus. The augmentation phase is enhanced to simulator a particular target architecture whose instruction set differs from that of the host. Application program is first compiled and optimized, as it would be on the target system. This intermediate code is then augmented with cycle counts. A second pass on the augmented intermediate code generates assembly code for execution on the host.

The above enhancement requires the compiler to be specifically modified for different target architecture. This is a major task calling for much time and effort. Therefore, the target architecture of the EPPP simulator is limited to only very few systems. It is the same as Tango, no in-session debugging tools are available in EPPP.

PARSE

Unlike Proteus or Tango, which uses a separate program to augment the compiled code, PARSE has code augmentation implemented directly in the compilation phase. The GNU C/C++ compiler was modified to augment parallel code when its basic block profiling flag is enabled.

This simulator is aimed at analyzing communication architectures and communication performance of parallel applications. Thus a high level of accuracy of code execution simulation is not of special interest to the simulator. For example, PARSE assumes that each instruction takes one clock cycle to execute and that memory accesses do not take additional cycles.

Concerning the debugging facilities, no tools are provided for correctness debugging of parallel programs, but performance debugging is available to analyze communication performance. However it is not user-friendly. User specifies the monitoring of various events performed within the communication network through a configuration file. The simulator will generate a trace file containing a time-sorted list of all requested events.

Detailed communication statistics can then be determined by examining these traces using data analysis tools.

Summary

Direct execution (both pure and with code augmentation) is fast but associated with two severe drawbacks:

- In-session debugging is very hard due to the nature of direct execution. Debugging and statistics rely heavily on the instrumented software approach. The accuracy of simulation results then depends on how much the monitoring code perturbs the system and application behaviors. The more traces/data required, the less accurate simulation results would be. This approach is thus not suitable for code developing or fine-tuning application performance.
- Simulation results are not accurate if the host's instruction set differs from that of the target. The inaccuracy also results from the fact that cycle counts of local blocks are accumulated at compile-time. In reality, execution time of an instruction depends on many run-time factors.

2.1.3 Functional Simulation

Functional simulation interprets instructions of the target machine at functional block level. This means each instruction of a target machine is expressed as macro or procedure whose size depends on the complexity of the instruction and the desired level of simulation accuracy. So, in functional simulation system, a parallel program is interpreted to intermediate object code. The set of intermediate object code should be definable to

match the target instruction set, and can be different from the host's assembly language. But it is not a necessary condition for accurate simulation because the simulator designer has control over how to interpret intermediate instruction. The level of detail of interpreting intermediate code determines simulation accuracy and time. There is a tradeoff between these two factors.

In order to get certain level of accuracy, functional simulation generally takes more simulation time than the direct execution approach. Even though it is a very attractive technique for performance debugging due to

- High accuracy: The high accuracy is gained due to the interpretation of intermediate instructions as if they were executed on the target machine. Monitoring code can be added to the simulating code without affecting simulation outcomes, because monitoring code can be distinguished from the application code and its execution time will not be accumulated into the total execution time. Cycle counts of intermediate instructions are accumulated at actual run-time (whereas direct execution simulators compute execution time of local blocks in advance at compile time).
- Flexibility: Code interpretation permits the simulator to have complete control over program execution. This allows establishing connection between user program code and intermediate instructions. User can set breakpoints, examine traced variables, or step through the program in a particular process. User can also view status of processes, processors and messages at any point during program execution.

Monitoring code can be added to the simulating code without affecting simulation outcomes, because execution time of monitoring code is not accumulated.

A typical simulator using functional simulation technique is Multi-Pascal simulator [17] [18] [29].

Multi-Pascal Simulator

This simulator simulates both shared-memory and message-passing architectures. User programs are first compiled into intermediate code. Each intermediate code instruction is associated with a fixed cost, which is the cycle count of that instruction on the target machine. At run time, intermediate code instructions are interpreted and their cycle counts are accumulated to give the total execution cost at the end of execution.

Multi-Pascal simulator provides a rich set of debugging tools, which benefit from the advantages of functional simulation technique. However, it still has some limitations, which prevent it from being an ideal performance debugger.

- In Multi-Pascal simulator, cycle counts of intermediate instructions are hard-coded into the interpreting code of the instructions and not well defined, therefore simulation results may not be accurate.
- The simulator does not support the concept of virtual architecture. The user needs to declare the physical architecture and specify the process-to-processor mapping in their application programs. There is no run-time mapping. Moreover, the user is forced to organize the program to match the available physical architecture that may

not be a natural structure for the application. This limitation makes study of program performance under different architectures inconvenient.

- The simulator assumes an underlying packet-switching network. There is no dynamic network simulation. Communication overheads of message passing are calculated based on one communication model. Again this makes the study of program performance under different network conditions inconvenient.
- The simulator does not support file I/O. It is only intended for small applications.

2.2 CPSS Simulation Technique

CPSS uses the functional simulation technique to simulate the execution of a parallel program on a multicomputer or multiprocessor system. Application programs are written in CPC language, which enhances the C language with parallel features to express process creation/termination and message passing. Similar to Multi-Pascal simulator, the CPSS also provides a rich set of debugging tools. In addition, it has some improvement over the Multi-Pascal simulator, which makes it an ideal performance debugger for studying parallel programming:

- Every intermediate code instruction is associated with a configurable cost, which can be adjusted to match a specific target machine and makes the simulation outcomes more accurate.
- Most of the computation and communication parameters are configurable. Values of the configurable parameters can be changed within the same simulation session as often as needed.

- CPSS supports virtual architecture programming and run-time process-to-processor mapping to improve programmability of message-passing applications. The user can write an application using a virtual architecture most natural to the application. At run-time the virtual architecture will be mapped to the available physical architecture. Moreover, the same source program can be mapped to different physical architectures without any changes to the source code. This makes the performance study under different physical architectures easy.
- CPSS contains a dynamic communication network simulator. It can simulate the commination network with four kinds of routing technique, Shortest Path, Simulated Packet, Packet Switch, and Wormhole Routing. CPSS can offer very accurate message routing and communication performance statistics for all routing models. Since wormhole routing technique is very popular on modern parallel computer systems, such as Intel Paragon [29], nCUBE6400 [30], and Ametek2010 [26], a parallel computer simulator that supports wormhole-routed networks has the practical value for performance study of parallel applications.

CPPE has a graphical user interface that eases the use of the rich set of debugging tools and the analysis of debugging information and program output

2.3 Routing Techniques

The existing routing techniques include packet switching, virtual cut-through, circuit switching, and wormhole routing [3] [4] [6] [7] [19] [20] [21] [23] [27]. In this

subsection, a concise description of these techniques and their characteristics are presented.

2.3.1 Packet Switching

This technique is also called store-and-forward switching. When a packet arrives at an intermediate node, the entire packet is stored in a packet buffer. The packet is then forwarded to the next node on the path where the next output channel is available and the next node has an available buffer.

Let P be the packet size (in bit), B the channel bandwidth (in bits/second), and D the distance between the source and destination nodes (number of hops). Ignoring message and packet startup overheads and packet blocked time due to resource shortage, the network latency incurred by one packet is (P/B)*D.

Packet switching is relatively simple to implement. It was widely used in first generation commercial multi-computers such as Intel iPSC/1, nCUBE/1, Ametek S/14, and FPS T-serials [26] [29] [30]. It becomes less popular because of the following drawbacks.

- Enormous buffering is required in every node.
- Routing overhead is occurred by the entire packet. The packet is buffered and retransmitted at every intermediate node on its path. It consumes too much time.
- The network latency is proportional to the distance between the source and the destination nodes.

2.3.2 Virtual Cut-Through

It can be considered as an enhancement to the packet switching technique. The enhancement is to reduce the amount of time spent on transmitting data by buffering the packet at an intermediate node only if the next required channel is busy.

Let P be the packet size (in bits), B the channel bandwidth (in bits/second), D the distance between the source and destination nodes (number of hops), and H the length of packet header (in bit) that stores control information such as the destination address and the packet sequence number. Therefore, the network latency is (H/B)*D + P/B. If H << P, the second term, P/B, will dominate the total latency and the distance D becomes a negligible effect on the communication latency.

If the network load is light the routing overhead (buffering and transmission) is significantly reduced comparing with packet switching. Packet routing latency is thus less sensitive to path length. However, if the network traffic is high, the routing overhead approaches that of packet switching, because the blocked packets must also be buffered.

This technique was adopted in the research prototype Harts, which was developed at University of Michigan. It is a hexagonal mesh multi-computer.

2.3.3 Circuit Switch

The routing of a packet in a circuit switched network involves three phases:

- Circuit establish phase: a physical circuit is constructed between the source and destination nodes
- Packet transmission phase: the packet is transmitted along the established circuit to the destination. During this phase, the channels constituting the circuit are reserved exclusively for this packet. Therefore, no buffering is required at intermediate nodes.
- Circuit termination phase: the circuit is released when the tail of the packet is transmitted.

Let P, B and D be the same as previous. Let C be the length of control data (in bits) that is transmitted to establish the circuit. Therefore, the network latency is (C/B)*D + P/B. If C << P, the second term, P/B, will dominate the total latency and the distance D has a negligible effect on the communication latency.

Second generation multi-computers such as iPSC/2 and iPSC/860 employ circuit switching as it has lower network latency and no buffer space requirement. However, it is very difficult for circuit switching to support sharing of physical links among contending packets. This results in low network utilization and susceptibility to deadlock.

2.3.4 Wormhole Routing

In a wormhole routed network, a packet is further divided into a number of flits (flow control digits) for transmission. Wormhole routing requires buffer at each node to store flits. As soon as a node examines the header flit of a packet, it selects the next link on the route and begins forwarding flits down to that link. The header flit governs the route.

While the header advances along the specified path, the remaining flits follow in a pipeline fashion. If the header flit encounters a link in use, it is blocked until the link becomes free. The flow control also blocks the following flits and they remain in buffers along the established path. When the last flit of the packet leaves a node, the link assigned to that packet is released and can be used by other packets.

Let F be the flit size (in bits). The network latency for wormhole routing is (F/B)*D + P/B. If F << P, the distance D will not affect the latency much unless the path is very long.

Wormhole routed networks allow virtual channels time-sharing a physical channel. As virtual channels occupied by a message are not released even if the message is currently blocked, this may lead to deadlock. Deadlock-free routing algorithms for wormhole routed networks have been proposed, which multiplex multiple virtual channels on the physical link.

Many multicomputer systems have used wormhole routing. Among them are Ametek 2010, nCUBE 6400, Intel/DARPA's Touchtone Delta, Intel Paragon, Intel/CMU's iWarp, and the Transputer IMS T9000 family. The popularity of wormhole routing is due to the following advantages:

- Network latency is relatively insensitive to path length.
- Required buffer space is small.

- It is easy to support virtual channels, which allow contending packets to time-sharing a physical link. Virtual channels help to improve network utilization, and implement deadlock-free routing algorithms in wormhole routed networks.
- Wormhole routing allows packet replication in which copies of a flit can be sent on multiple output channels (virtual channels). Packet replication is useful in supporting broadcast and multicast communication.

2.4 CPSS Communication Network Simulator

In order to study how different routing techniques have influence on message passing process, our simulator currently deploys the routing techniques described above: packet switch and wormhole routed technique. In addition, another two types of routing techniques are also implemented in the early time when developing the parallel system simulation. They are shortest path and simulated packet.

Chapter 3

CPSS System Architecture

In this chapter, the design objectives of CPSS are discussed. By analysis how to match these objectives, the high level design of CPSS and the design criteria for communication network simulator is presented.

3.1 Design Objectives

The most important objectives considered in the design of CPSS are as following.

1. Realistic modeling: CPSS should reflect accurately the behaviors of the multicomputer system as well as the performance of an application program on this system. So the design should follow realistic computation and communication models, and retain the essential characteristics of the multi-computer system.
2. Accurate simulation: CPSS should employ the functional simulation approach to simulate the execution of parallel programs. At the same time, it should execute an application and simulate the behaviors of the underline multi-computer system.
3. Performance: Time of simulating the execution of application programs is also a crucial issue. The granularity of instruction interpretation should be weighted carefully to meet both the accuracy and performance requirements.
4. Flexibility: CPSS should offer user the freedom of changing system parameters for both computation and communication. This enable user to tune an application

to the underlying hardware at hand, or to obtain a though performance analysis of the application on various multicomputer systems which have different characteristics. The flexibility of changing parameters should also come with convenience: the parallel program need not be re-compiled every time when system parameters are changed.

5. Repeatability: Repeatability is necessary to study different phenomena in an execution at several levels of detail and from different perspectives. It is essential to provide a stable and reliable debugging environment that is not available on real multicomputer systems. Real parallel computers are nondeterministic in nature, and rarely provide any form of repeatability. In this case, some bugs may not occur frequently enough for observation. Repeatability does not mean that the simulator can reproduce only one of the many possible executions of a nondeterministic application. The simulator should also be able to mimic the nondeterministic nature of real multicomputer systems and of parallel applications.
6. Correctness debugging: CPSS should provide convenient debugging tools and useful debugging information to end-users in order to test and debug application programs. This is the main advantage of using a simulator rather than a real multicomputer system. The debugging code embedded within the simulating code should not affect behaviors of application programs and their outputs.
7. Performance debugging: Computation and communication statistics of application execution should be provided to facilitate the parallel architecture, network characteristics, parallel algorithms and program mapping. Global statistics of an

application (such as total execution time, speedup, and total computation and communication time) allow user to tune the application to a desired performance. Run time data or traces (such as process creation/termination information, or message send/receive information) could be very useful in studying a particular aspect of the application, which cannot be captured by global statistics.

8. User-friendliness and portability: It should be easy to learn and use the simulator. An intended use of the simulator is to introduce the field of parallel computing to new learners. Also, the simulator should be potable to various platforms.

3.2 System Architecture

As communication network simulator is an important part in CPSS, it works together with other modules in CPSS to complete the parallel computer simulation. Before we go to the design of the communication network simulator, we need to present the high level design of CPSS, which provides the foundation for the design and implementation of CPSS communication network simulator. At the same time, we will see how the parallel computer simulator design objectives listed in the previous subsection are satisfied. At the end, the high level design of the communication network simulator is presented.

3.2.1 General Structure of CPSS

In order to meet the design objective 1, CPSS must consist of two major components: the code execution module and the communication network module. The code execution module models the processing elements of the parallel computer system. It executes the parallel code specified by the parallel program. This module determines how much the

design objective 1, 2, 3, and 4 are satisfied. The communication network module is to manage the inter-processor communication via message passing, and it affects the feasibility of design objective 2, 3, 4, and 5. This module is the main topic in this thesis and also called communication network simulator. There are two other utility modules interacting with the code execution module and the communication network module in CPSS, the debugging monitor and the user interface, to meet the other design objective 4, 6, 7, 8.

The general structure of CPSS can be described as following:

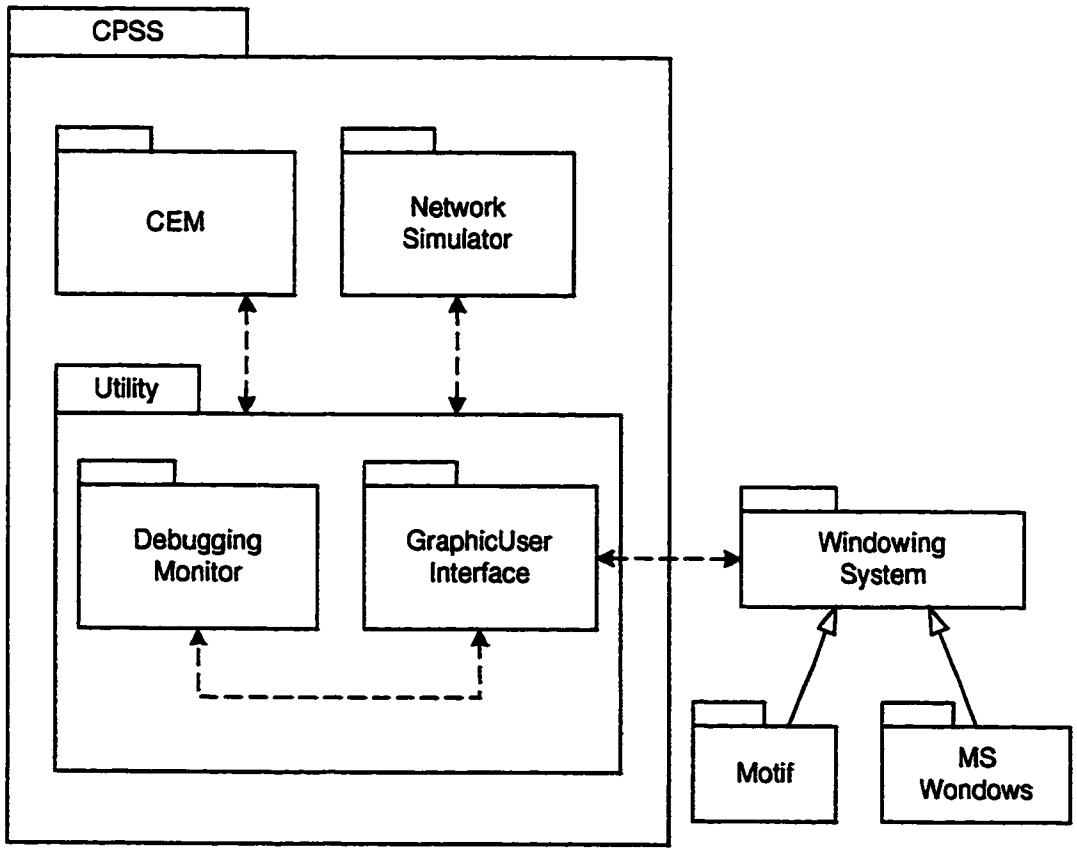

Figure 3: General Structure of CPSS

3.2.2 The Code Execution Module

The code execution module (CEM) plays the role of processing element of a parallel computer system. It executes the parallel code specified by the parallel program. The major entities in CEM are processing elements (processors), local memories of processors, processes, and communication channels among processes. The internal structure of CEM and the relationship between each other are described in Figure 4.

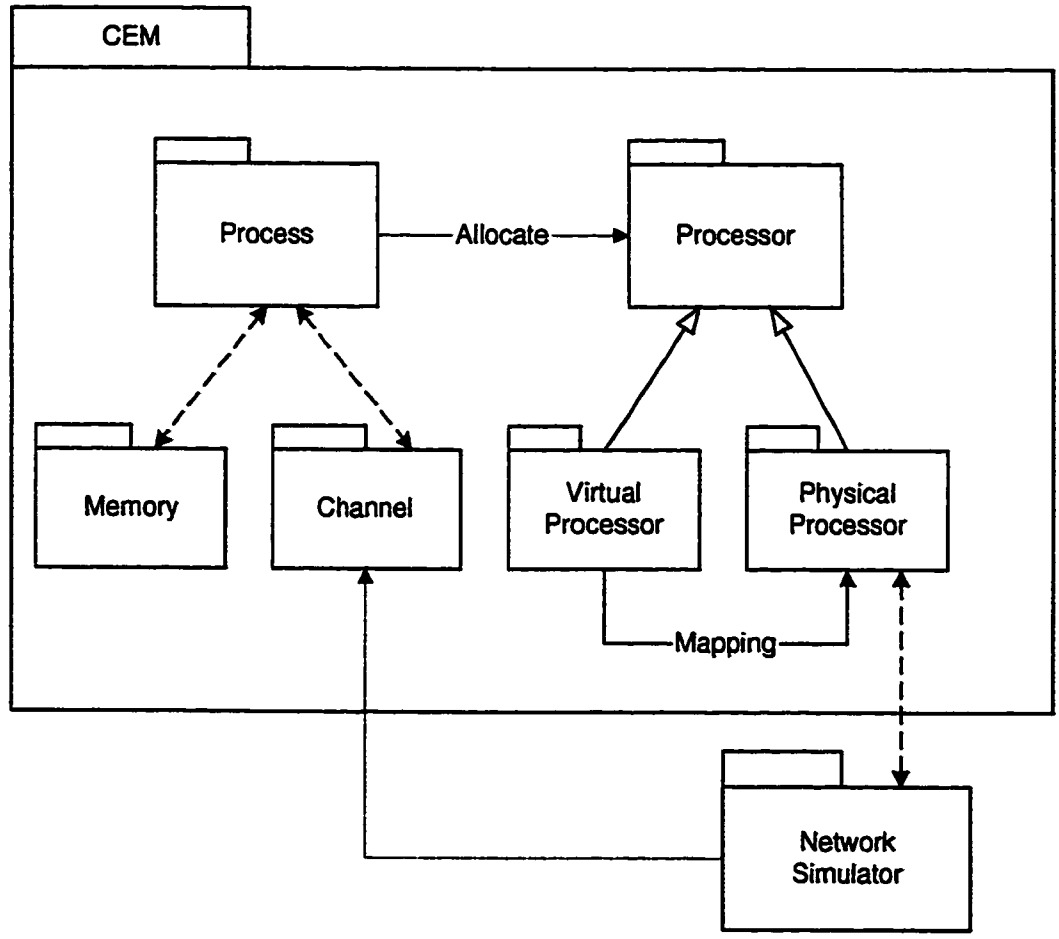

Figure 4: CEM Structure

The CEM maintains a global clock for the simulated multicomputer system, which is updated periodically by the CEM. There are four major functional parts in CEM to coordinate all entities.

1. Map Manager: The map manager maps virtual processors specified in a CPC program to physical processors. CPSS provides default mapping whose objective is to minimize the path contention level of the parallel program. There are also identical mapping, random mapping available. In addition, users can define the mapping by themselves.
2. Storage Manager: The job of the storage manager is to allocate simulated local memories to processes upon process creation and deallocate this space upon process termination. The storage manager also allocates/deallocates other kinds of dynamic memory blocks such as activation records upon function calls/returns, and buffers for messages of composite types.
3. Process Scheduler: The process scheduler schedules processes for execution and updates their structures according to changes in process and processor status, local clocks and the global clock. In CPSS, parallelism is simulated by time slicing. Each application process is given a quantum to run and processes are scheduled in a round-robin manner. During each quantum, the process scheduler traverses the list of processes, and schedules one process at a time for execution. The process runs until its quantum expires or it is put to sleep by some event. After each application process has finished its quantum, the global clock is advanced to the next quantum.

4. Instruction Interpreting Routines: These routines interpret vCode instructions. Each instruction is associated with a cost, which the routine will look up in a cycle-count table to update the local clock of the current process accordingly.

3.2.3 The Debugging Monitor Module

The debugging monitor is responsible for handling the debugging mechanisms. The CPSS system architecture provides a good debugging environment, which is inherited from the functional simulation technique and the timing system employed in the design of code execution module and communication network simulator. CPSS provides two types of debugging tools, the correctness debugging and performance debugging.

3.2.3.1 Correctness Debugging Tool

As parallel object code instructions are interpreted at the functional level, it is convenient to insert debugging code inside the interpretation code as much as we need to implement the necessary debugging functions. From chapter 2, we already know that insertion of these debugging code does not affect the simulation results in terms of execution cost since the debugging code can be distinguished from application code.

CPSS supports the following debugging tools:

- Set and clear breakpoints in user's parallel program source code.
- Set and clear trace variables, whenever the variable is referenced, the program execution is suspended.
- View a value of a variable in an active process when execution is suspended.

- Step through a suspended program line by line or by a specified number of lines.
- Set a particular process to be the current process for debugging.
- Display user's parallel program source code with specified range.
- List the vCode corresponding to the specified range of the source code.
- View the status of the processes, including the processor on which a process is running, the current execution status of a process...

3.2.3.2 Performance Debugging Tool

The performance debugging tool not only needs to provide the overall performance profile of a parallel program execution, but also to provide the functionality to study the performance of any portion of the parallel program. As the performance degrade sources are in parallel computation, parallel algorithms, and communication, the following performance debugging tools are provided in CPSS:

- Set and clear time breakpoints by using an alarm to automatically suspend the program execution when a certain time is reached.
- Provide parallel execution time, which is the estimated execution time on an actual parallel computer.
- Provide sequential execution time, which is the estimated execution time on a uniprocessor computer.
- Provide execution time of any portion of the program, either sequential or in parallel.
- Provide computation time, which is the time spent on communication task.
- Provide communication time, which includes overhead involved in inter-processor communication, such as message sending/receiving, congestion delay.

- View processor utilization.
- View profile of processor utilization as a function of time.
- List process creation/termination information, such as time, processor number, parent process ID.
- List message sending/receive information, such as time, source node, destination node, and message length.
- List message routing information, such as path, time traces.

3.2.3.3 System Configuration

The debugging monitor module not only provides the tools listed above, but also provides the configuration management tool. The following computation and communication parameters can be configured through debugging monitor module:

- Physical architectures
- Mapping functions
- Routing techniques
- Network parameters

3.2.4 The User Interface Module

The design purpose of user interface is to enable user to interactively communicate with CPSS. The user interface receives parameters and commands from user, validates the received information, and passes them to the appropriate module (CEM, communication network simulator, and debugging monitor module). During execution of a parallel program, the user interface interacts with debugging monitor to display performance

statistics and debugging information. Program output is also transferred from the CEM to the user interface.

Due to the complexity and multiple dimensionality of parallel performance data, graphic user interface (GUI) is provided both under Unix system and Windows environment to help parallel programmers in their development process. The GUI under Unix is developed with Motif toolkit, and the GUI under Windows is developed with Microsoft Foundation Classes (MFC).

Figure 5 shows the graphic user interface under Windows. On the bottom of the window, debugging tools are listed.

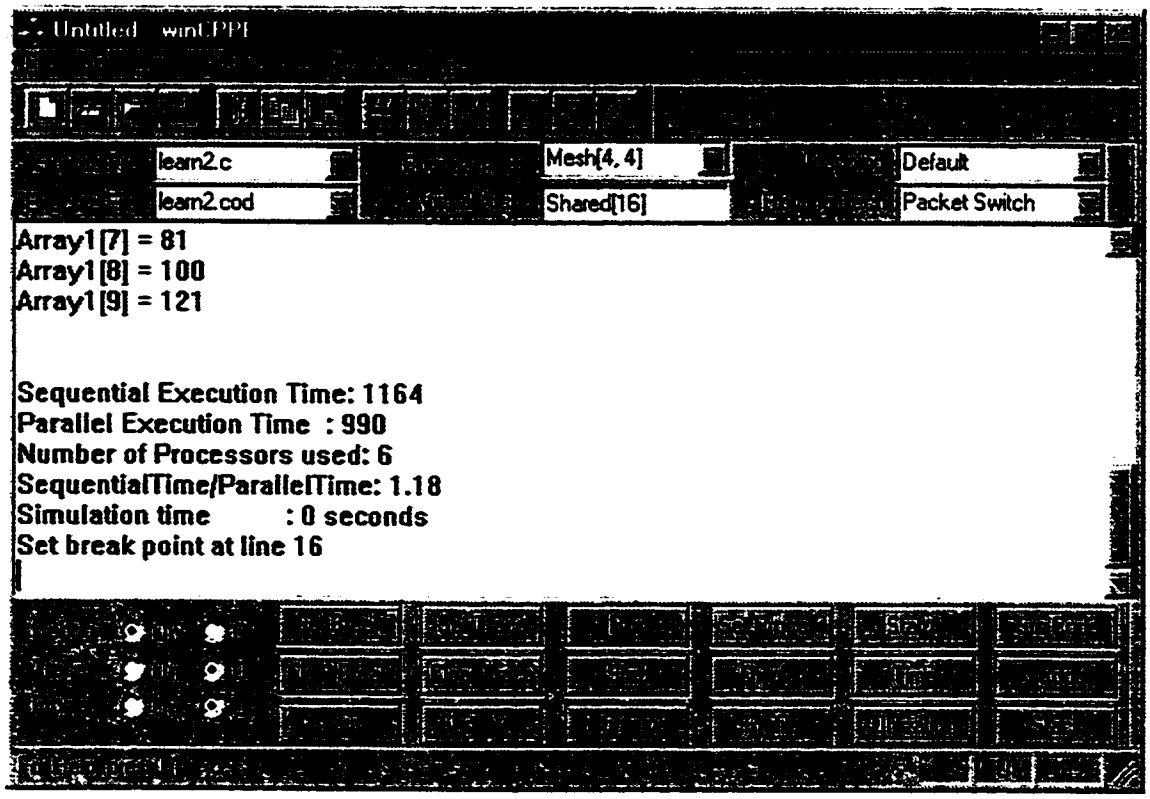

Figure 5: GUI under Windows system

Figure 6 shows the graphic user interface under Unix system:

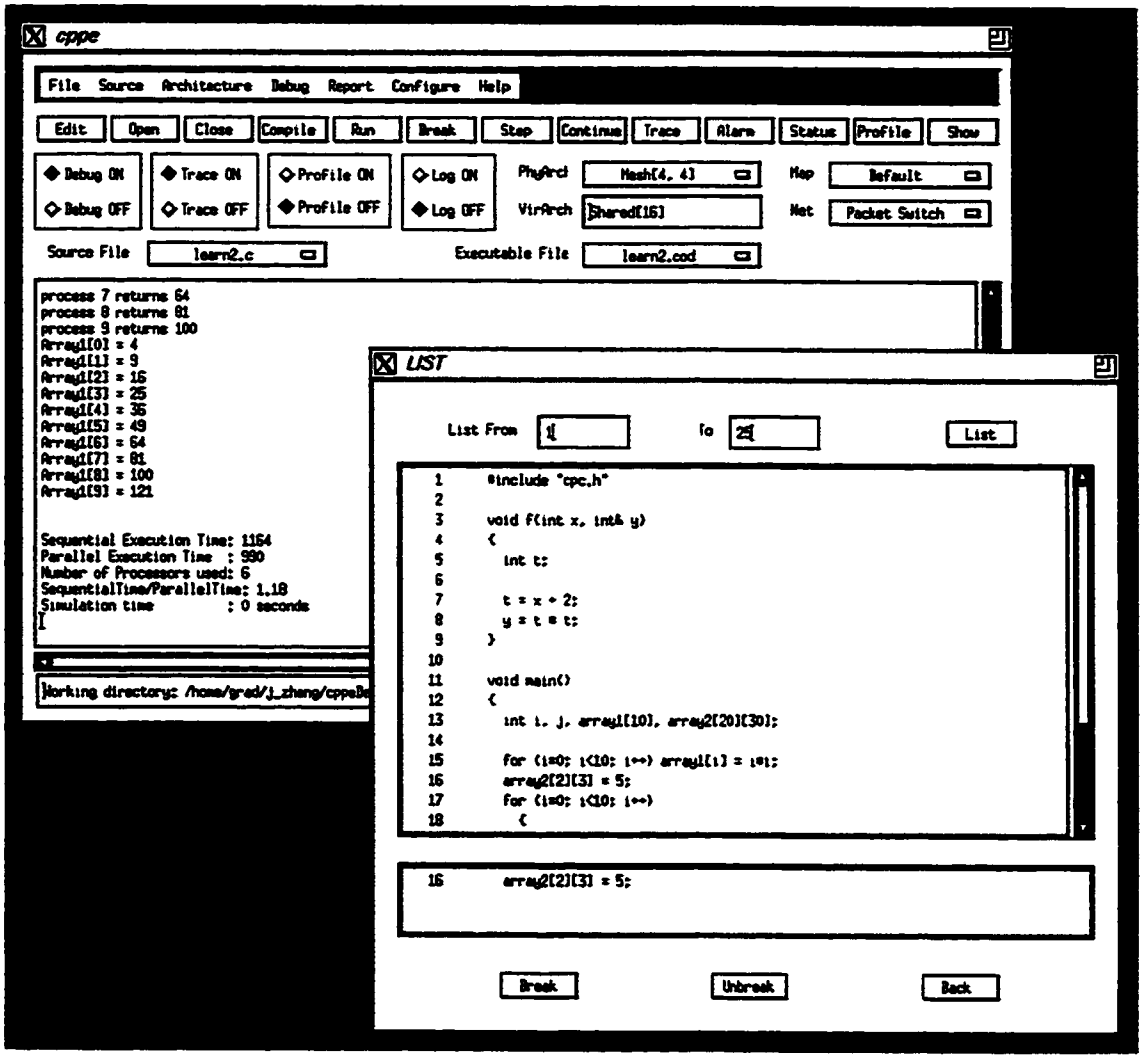

Figure 6: GUI under Unix system

Both the Unix GUI and Windows GUI are isolated from the other components in CPPE in order to achieve certain level of modularization. Since CPPE is still under development, modularization helps to de-couple and therefore reduce the dependence between CPPE modules. To support portability, compilation condition flags are used to adapt the simulator to different development environment.

3.3 High Level Design for Communication Network Simulator

The communication network simulator is responsible for inter-processor communication via message passing in CPSS. It is under control of the network manager. The network manager allocates network resources to messages to be sent, routes messages, delivers them to destination processors, and detects and resolves deadlock, if any.

The communication network simulator is tightly linked with the system underline topology and the routing technique. CPSS supports the following typical network topologies: line, ring, mesh, torus, and hypercube. The higher the network topology dimension is, the lower the network latency is, but the more difficult the implementation is. CPSS provides the network topology dimension from one dimension (line, ring) to three dimensions (hypercube) for user to further understand the relationship between network latency and system topology.

In CPSS communication network simulator, four types of routing techniques are implemented. They are shortest path, simulated packet, packet switch, and wormhole routing.

- Shortest path is the simplest, but the least accurate method. Only the distance between the source node and destination node is considered when estimating the network latency. It is used at the early time when CPSS was developed to roughly estimate the network latency.

- Simulated packet is based on shortest path, but the network congestion is considered. In the shortest path and simulated packet, it is assumed that the whole message is forwarded altogether.
- Packet switch is designed to both remove the instability of transit time with increasing load, and prevent the delays caused by sending long messages over limited capacity of communication channels. This is achieved by splitting messages up into small fixed size pieces, called packets, each with the destination address contained therein. In network using dynamic routing, the packets are also given a sequence number so that they can be reassembled into coherence messages, even if the constituent packets making up a message have been sent by different routes and have taken different time to reach their destination. It is more complicated, but it is more accurate. In our simulator, a message is not really forwarded, only the network latency is calculated.
- Wormhole routing is similar to packet switch, but it is more complicated and deadlock may occur. Instead of passing packets, a packet is further divided into flits, and a flit is forwarded each time. When a header flit reaches an intermediate node, it can request a new route to continue forwarding further without waiting for the whole flits in a packet arriving. By doing so, a pipeline is formed and the network latency is reduced significantly. The degree of parallelism is increased. But at the same time, the condition of deadlock could be created.

The communication network simulator should contain the following entities to performance the message passing: system topology, routing technique and message (packet, flit). Figure 7 illustrates the structure of communication network simulator.

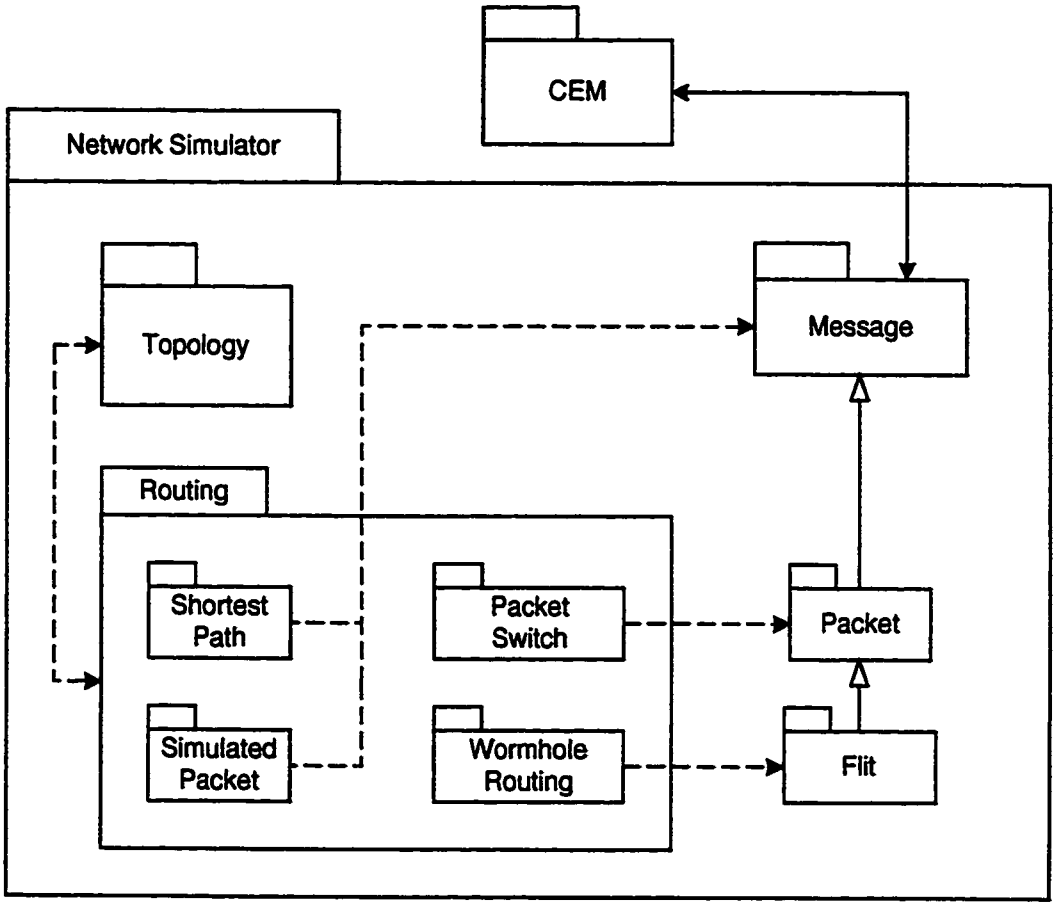

Figure 7: Structure of Communication Network Simulator

When a message needs to be sent, the sender process writes the content of the message to a channel variable (see section 4.1.1 for the description of channel variable), invokes routine netSendMessage() to pass message information to communication network simulator. Then the sender continues with its execution. The communication network simulator will simulate the message route procedure using the information received from

the sender. When a message reaches its destination, the communication network simulator gets the network latency and notifies CEM of the arrival of the message and performs corresponding operations according to the message type. Then the intended reader can read the message from the channel variable. The design and implementation details of communication network simulator will be presented in the next two chapters.

3.4 Measures to Meet the Design Objectives

3.4.1 Accurate Simulation

CPSS uses the functional simulation approach to simulate the execution of parallel programs on a multicomputer system. The level of detail of code interpretation decides the accuracy and the performance of the simulation. The tradeoff between accuracy and performance has been considered carefully in the design of the CPSS to provide accurate simulation result within an acceptable amount of running time. The degree of accuracy is adjustable and decided by the definition of vCode instructions, their associated costs, and their level of detail of interpreting each instruction.

Accurate simulation of parallel architecture is also promoted by the ability that most computation and communication parameters are configurable. These configurable parameters allow user to accurately simulate a particular multicomputer system.

3.4.2 Performance

Although CPSS uses the functional simulation technique, the simulation is not done at the machine instruction level in order to speedup simulation time.

We analyzed basic operations of existing parallel architectures and constructed a set of parallel primitives, which are common to most target multicomputer systems. Parallel primitives are simulated at the functional level with a reasonable abstraction to tradeoff between simulation accuracy and simulation time.

The entire simulation system, including the application program, is run by a single process. The simulation does not have any context switching on the host, and thus saves simulation time. Further more, the network simulation does not route actual messages. The routing is simulated using only message information. In addition, the network uses an approximate version of round robin scheduling to speedup the simulation time. Thus, high performance is achieved.

3.4.3 Flexibility

Besides accuracy, the functional simulation technique also deploys the other simulation techniques in terms of flexibility and debugging convenience. The CPSS offers flexibility to users in many ways.

Virtual to physical architecture mapping at run time

The application program is written using the virtual architecture, which should be the most, and efficient architecture for the application. At run time, the virtual architecture will be mapped to a physical architecture. If the user does not specify any physical

architecture, the default is the virtual architecture itself. In this case, no mapping is needed.

CPSS provides the following mapping type:

- Default mapping: The objective of this mapping is to minimize the path contention level of the parallel program.
- Identical mapping: The mapping function maps the virtual topology to the same topology on physical processors.
- Random mapping: The system maps the virtual topology to a randomly picked physical architecture.
- User defined mapping: Users can configure the mapping by themselves.

Simulating a wide range of multicomputer systems

Unlike direct execution simulators, a functional simulator can be adapted to simulate a new architecture more easily because the intermediate instruction set can be expended or re-defined in terms of instruction functionality.

- Simulate different target machine: The vCode instruction set of the CPSS is constructed using common operations of multicomputer systems. The cost of each instruction can be adjusted to simulate a specific target machine. Instruction functionality can be modified and new instructions can be added easily due to the modular and decoupled implementation of the simulator.
- Simulate different system topology: The CPSS can simulate a wide range of topologies, including line, ring, mesh, torus and hypercube. System size can be

extended up to thousands of nodes and is limited only by the memory capacity of the host computer.

- Simulate different communication routing type: Multiple communication routing types can be simulated by CPSS, including shortest path, simulated packet, packet switch and wormhole routing. As object oriented design is applied to the communication module, new routing techniques can be added easily with affecting other parts of the simulator.

Large set of configurable parameters

User can configure almost all computation and communication parameters. These changes can be done in the same simulation session and no recompilation is needed. That means the same vCode of the application program is executed. This feature also enables the repeatability. Tuning system parameters allows the analysis of an application or architecture to be more thorough and accurate.

Modularity and expendability

The design and implementation of the simulator are modular and decoupled. Future changes and enhancements to the simulator would be quick and easy. For example, if another type of routing technique is required to be added to the simulator, a class of that routing type is declared, which is inherited from network base class (CNetwork class), implements only the virtual functions defined in CNetwork, which respects the specification of this routing type.

3.4.4 Repeatability

The repeatability is easily achieved since the simulator is a sequential program, which is deterministic and repeatable in nature. Under the same setting of parameters, different executions of the same application yield the same results.

Repeatability does not imply that only one of the many possible executions of an application can be simulated. In fact, the simulator can reproduce multiple executions of a nondeterminic application. This is particular useful for applications whose behavior is considerably different from one run to another depending on the outcome of race condition.

In CPSS, multiple executions are implemented by varying the relative processor speed. Race conditions can thus be created by variation of relative processor speeds. For example, by changing the relative processor speeds, the order of sending two messages from two distinct nodes may be reversed.

Repeatability is critical to provide a stable and reliable debugging environment. It also helps to study an application at various levels of details and from different angles. Multiple executions are essential to study nondeterministic applications, which demonstrate different behaviors depending on results of race conditions.

3.4.5 Correctness Debugging

Another advantage of the functional simulation over other simulation technique is easy debugging. Since parallel object code instructions are interpreted at functional level, it is

easy to insert debugging code inside the interpretation code and it does not affect simulation result.

The correctness-debugging tool provided by CPSS, listed in 3.2.3.1, shows that the design objective for correctness debugging is satisfied.

3.4.6 Performance Debugging

The performance data and statistics produced by CPSS are listed in 3.2.3.2. These data records are logged into files and can be enabled/disabled as user wishes. It is also easy for users to add their own traces to the simulator code in order to capture performance data as they want.

3.4.7 User Friendliness and Portability

A graphic user interface (GUI) is used to integrate the major components in CPPE (CPCC and CPSS) into a unified and user-friendly parallel programming environment. Through interaction with the GUI, user can accomplish the whole development process from source code editing, to compile, debugging, execution, and performance profiling.

The GUI module is designed to make program developing and debugging easy for the parallel application programmer. At the same time, the GUI can be easily configured to run in different environments and on different platforms. In the current version of CPSS, there are two GUI modules are provided. One is for Microsoft Windows environment, and another is for Unix environment.

Chapter 4

The Design of CPSS Network Simulator

By analyzing the purpose and the feasibility of communication network simulator, the design objectives are summarized. Then the design of the communication network simulator, which is to achieve these objectives, is presented.

4.1 Network Overview

The purpose of the communication network simulator in CPSS is to simulate message passing procedure between processors in order to get network latency. With network latency, the destination processor knows when the message is ready to be picked up. In our simulator, messages are not actually routed through network nodes. They are stored in channel variables with big available time when they are sent. After the network latency is achieved, the message's available time is updated. As soon as the current clock reaches the message's available time, the message is ready to be retrieved by the receiver.

Here, a new terminology, channel variable, is introduced. Since messages are passing through channel variable, the channel related parameters are described below.

4.1.1 Channel Related Parameters

Channel variable is used to store message contents. It has the following data structure:

value	*time*	*size*	*processID*	*tag*	*next*

Where:

- *value*: value refers to the message contents. It can be the value for basic types *basicValue* or address for basic type. In cpss, *basicValue* can be integer, float, or composed type
- *time* is message arrival time
- *size* is message size
- *processID* is message destination process
- *tag* is user tag for message, can be thread ID
- *next* is link the channel value node together

Channel variables can be wrote by any process, but it can be read only by its owner. Therefore, each channel variable is associated with another data structure which indicates the owner of the channel variable. It has the feature of mailbox.

There is another parameter called channel descriptor, which holds the control information of channel. The data structure for channel descriptor can be described as following:

head	*dataCount*	*waitProcQueue*	*earliestReadTime*	*channElemSize*	*link*

Where:

- *head*: a channel is considered to be an infinite buffer of messages of the same type. Messages written to a channel are stored in the increasing order of the

writing times, and maintained in a list for reading by the channel owner. This list will be referred to as the list of channel value, *head* points to the first of this element of this list.

- *dataCount*: number of channel values available in channel
- *waitProcQueue* is list of waiting processes for channel read
- *earliestReadTime* is earliest time the channel can be read again
- *channElemSize* is the size of messages written to this channel
- *link* is index to next available channel

4.1.2 Message Passing Procedure

The message passing procedure is described in Figure 8.

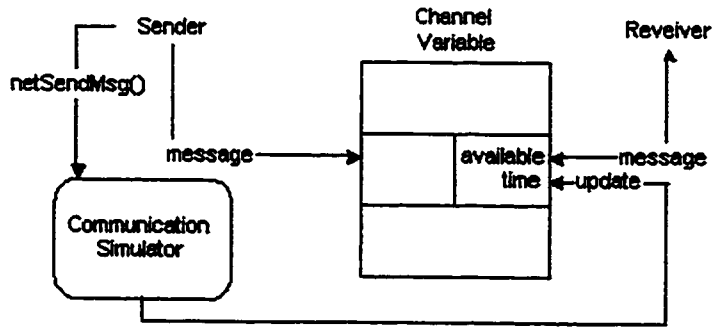

Figure 8. Overview of message passing procedure

4.2 Consideration on Network Simulator Design

Message passing event is invoked by process. If the data exchanged between two processes running on the same processor, the network latency is the local message cost. It does not invoke the real message passing procedure and it is not the major concern of the design our network simulator. Our major concern is based on the message passing

procedure happened between different processors. In this subsection, we first study the affecting factor on network latency. By analyzing the code execution module in CPSS, we list the situations when message passing occurs. Finally the message type is summarized.

4.2.1 Affecting Factors on Network Latency

The purpose of CPSS communication network simulator is to simulate the real message passing procedure in order to get the accurate message delay time. There are two major factors which have influence on this delay time. They are system topology and routing technique.

System topology determines how multiprocessors are organized in CPSS. It has direct influence on the network latency. Comparing with a 8-node line topology with a 8-node ring topology. If a message is sent from node 0 to node 7, in line topology, the message has to pass all nodes to reach the destination, but in ring topology, the message can go through another direction and goes to node 7 directly. Therefore, the system topology is the main concern in the design of network simulator.

Routing technique determines how messages are routed. Messages can be routed as a whole. It can be divided into packets and the network simulator routes the packets instead. In addition, packet can be further divided into flits and flits are routed inside network simulator. These different ways of message forwarding form the different routing techniques.

In CPSS network simulator, four types of routing techniques are supported. They are shortest path, simulated packet, packet switch and wormhole routing. Besides the difference of routing techniques causes the difference on message delay time, the underlying system topology makes this difference more significant. With shortest path and simulated packet routing techniques, the following network topologies are simulated: line, ring, mesh, torus and hypercube. With packet switch and wormhole routing techniques, the following network topologies are simulated: line, ring, 2D-mesh, 2D-torus, 3D-mesh, 3D-torus and hypercube. Only low dimension mesh and torus are implemented, in order to simplify the implementation.

4.2.2 Situations When Message Passing Is Invoked

In CPSS, message passing occurs when

1. Child process needs to write back to its parent process
2. Perform VIRTUAL_SEND and PHYSICAL_SEND.
3. Perform VIRTUAL_BROADCAST or PHYSICAL_BROADCAST
4. Perform COPYBLOCK between processes
5. Store value from stack to channel STCHANNEL
6. Perform NEWFORKCHILD or NEWFORALLCHILD operation
7. Perform FORKCHILDEND or FORALLCHILDEND operation
8. Perform WAKEUP operation

4.2.3 Message Type

By analyzing the situations and purposes when messages are sent, messages can be grouped into the following types:

- IGNORE_ME (case 1, 4 listed above)
- SEND_DATA (case 2, 3, 5 listed above)
- BIRTH (case 6, 8 listed above)
- DEATH (case 7 listed above)

IGNORE_ME is a kind of message that is sent to cause traffic in network system. After the message arrives at the destination process, this message is ignored and no further operation is invoked.

SEND_DATA is a kind of message that is really sending data to the destination process. After the message arrives at the destination, the message is inserted into channel queue by calling insertValueInChann() function. When the destination process need to read the data, it queries the channel queue and retrieves the data if its arrival time is earlier than current time. As soon as the data is retrieved, the message is removed from the channel queue.

BIRTH is a kind of message that is sent to wakeup new process. After a message arrives at the destination process, it wakes up the process by calling wakeNewProcess() function provided by CEM.

DEATH is a kind of message that is sent to terminate the child process. After the message arrives the destination, it terminates the process by calling deathProcessing() function provided by CEM.

4.3 Network Simulator Design Objectives

The design objectives of the CPSS communication network simulator:

- Encapsulation: Since the network latency depends on the system topology and routing technique, and the code execution module does not affected by these factors. Therefore the communication network simulator module can be extracted from code execution module, and all network related information and implementation details are hidden from CEM.
- Polymorphism: Polymorphism is an important feature of inherence. It allows two or more objects in an inherence hierarchy to have identical member functions that perform distinct tasks. The computer environment selects a version of the function appropriate to the situation. As our network module supports four types of routing techniques, without this feature, CEM has to recognize which routing technique the simulator is using and call the corresponding send message function. At this time, the insulation can not be ensured. With polymorphism, the network module only exposes its abstract interface to CEM and CEM does not need to aware which routing technique and what topology the system is running on.
- Easy extension: There are four types of routing techniques are supported in our current release of CPSS. Probably there will be a request to support more routing techniques in the future. With the two characteristics listed above, this feature is easy

to be achieved. Programmers do not need to go through the whole CPSS code to do modifications, and only the network simulator module needs to be accessed.

4.4 Design of Network Simulator

By analyzing the communication network simulator design objectives, we realize that object oriented design can encapsulate the network module from other parts of CPSS, insulation can be achieved. By using an abstract class as network interface, other modules in CPSS makes the same function call and does not need to know which routing technique is applied, the right object is automatically linked. The polymorphism objective is achieved. With this design, future network extension becomes much easy. If another routing technique needs to be supported, we simply derive a new class from the network abstract class and implement this class independently.

4.4.1 Network Simulator Structure

Figure 9 shows the class structure of communication network simulator[11][24][28]. As our current network simulator supports four types of routing technique, five classes are declared. They are CNetwork, CShortestPath, CSimulatedPacket, CPacketSwitch, CWormhole.

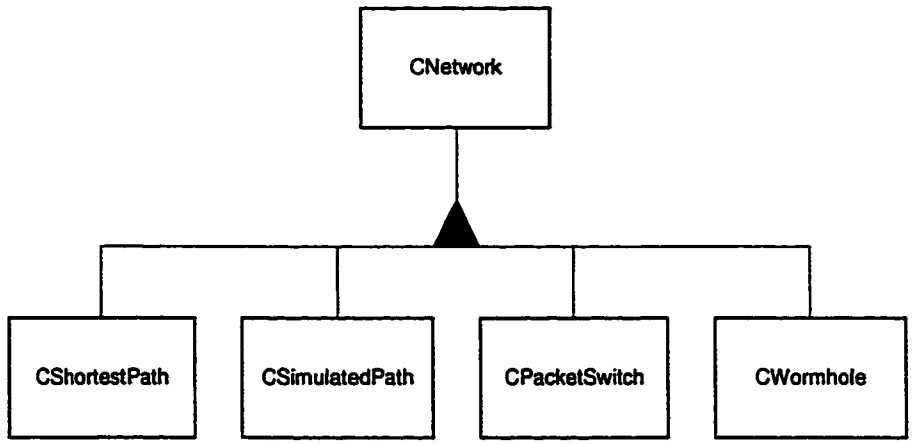

Figure 9: Class Structure of Communication Network Simulator

CNetwork class is a pure abstract class. It defines all interface functions the network simulator provides. All other classes are derived from it and they have their own version of interface functions.

The CPSS network simulator is referenced through a CNetwork type pointer *pNetwork*. It is declared globally and can be accessed throughout CPSS. According to the current selection of routing technique, an object of that type is created and is referenced by *pNetwork*. Whenever the network type is changed, the current object is deleted and a new type of network object is created. Figure 10 illustrates how a CNetwork type pointer is declared and used:

```
// Declare a network pointer
CNetwork *pNetwork = 0;

// default network type is shortest path
pNetwork = new CShortestPath;
      .
      .
      .
```

```
// usage of CNetwork pointer
// the netSendMsg() function is the version in CShortestPath
pNetwork->netSendMsg( … );
      .
      .
      .
delete pNetwork;
```

Figure 10. The Usage of CNetwork Type Pointer

Through this network pointer, the right version of network related functions are linked automatically. That's the polymorphism feature of C++ language. The fact that CPSS always links to the right version of the network object can be guaranteed.

4.4.2 Network Simulator Interface

The network simulator exposes interfaces that CPSS required during message passing procedure. By analysis the functionality of the network simulator, the interface functions can be divided into the following groups:

- Network initialization: Allocate, deallocate, or reset network resources to CPSS. Functions in this group are called during the system initialization time or when a new type of routing technique is selected. It includes the following methods: class constructor and destructor, restart a new run (*INIT_NEW_RUN*) method.
- Network parameter configuration: Functions in this group provide facilities to set network related parameters, the network topology, the definitions for the number of network lane/link, packet size, buffer size, and time delay for message startup. Functions in this group are called through user interface. It includes the following methods: *INIT_CHANGE_PHYS_ARCH*, *PARMS_CHANGE*, *PRINT_PARMS*.

- Message sending: It initializes the message passing procedure and finally gets the network latency for the message. Then it informs the destination process about the arrival of the new message, and according to the type of message, performs the corresponding operation. The functionality is performed by the method: *netSendMsg*.
- Network routing schedule: For shortest path and simulated packet routing technique, the message delay time can be calculated directly by counting number of hops between the source and destination nodes. This is done in message sending procedure and does not have routing schedule. But for packet switch and wormhole routing techniques, the calculation for the network latency is more complicated. The message delay time can not be got immediately. The network has to forward the message one node by one node till its destination. So, message-sending operation, in this case, is just inserting the message into message queue. When network get the time slice for execution, it performs the message forwarding operation at this time. Therefore, the message passing procedure is executed at this time. This functionality is performed by method: *ROUTING_ROUND*.
- Deadlock handling: Deadlock happens when a process holds a resource, say resource1, and requests another resource, say resource2, and at the same time, another process holds resource2 and requests resource1. Before the requested resource is acquired, the holding resource can not be released. In this case, deadlock happens. Wormhole routed network has potential situation which can cause deadlock happen. In wormhole routed network, the sending message is divided into packets, and a packet is further divided into flits. Finally, a flit becomes the smallest unit forwarded through network. The advantage of wormhole routed network is that the header flit

can continue be forwarded without waiting for the arrival of the end flit. A message can form a pipeline and speedup transmission speed. On the other hand, as the header flit requests a new lane while the other flits still on the old lane, this has the potential to cause deadlock. For the other routing techniques, since there is no pipeline situation, deadlock can not happen on them. Therefore, only wormhole-routed network has the implementation for deadlock handling methods. They are *netDeadlockRecovery*, *networkDeadlock* and *netDeadlockReport*.

In order to apply polymorphism, the CNetwork class exposes all interfaces, which is needed by all types of routing technique. All other classes inherence from it and keep the same interface as it has, but have their own version of implementation. Figure 11 shows the class definition for CNetwork class. Some of the initialization functions are included in class constructor and destructor.

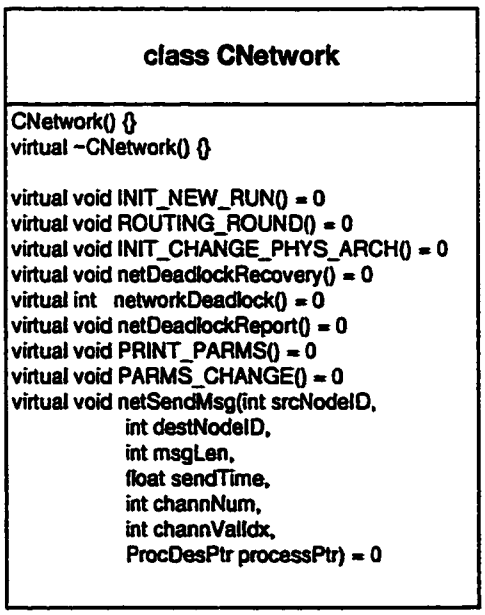

Figure 11: Class CNetwork Definition

4.5 Working Environment for Network Simulator

In the previous section, the class definition for CNetwork has been presented. The propose of exposing these interface methods is to encapsulate the data structures and implementation details of the network simulator from rest of the simulation system. The network simulator is an important part in our CPSS and requires coordinating with the other modules to complete the parallel simulation. In this section, the working environment for network simulator and the interactions between network simulator and the rest of the system are presented.

We already know that CPSS contains three main modules, they are Code Execution module, Graphic User Interface module and Communication Network module. The interface functions defined in Figure 11 are windows for other parts of CPSS to access communication network simulator. The following diagram, Figure 12, illustrates where and when the interface functions are invoked.

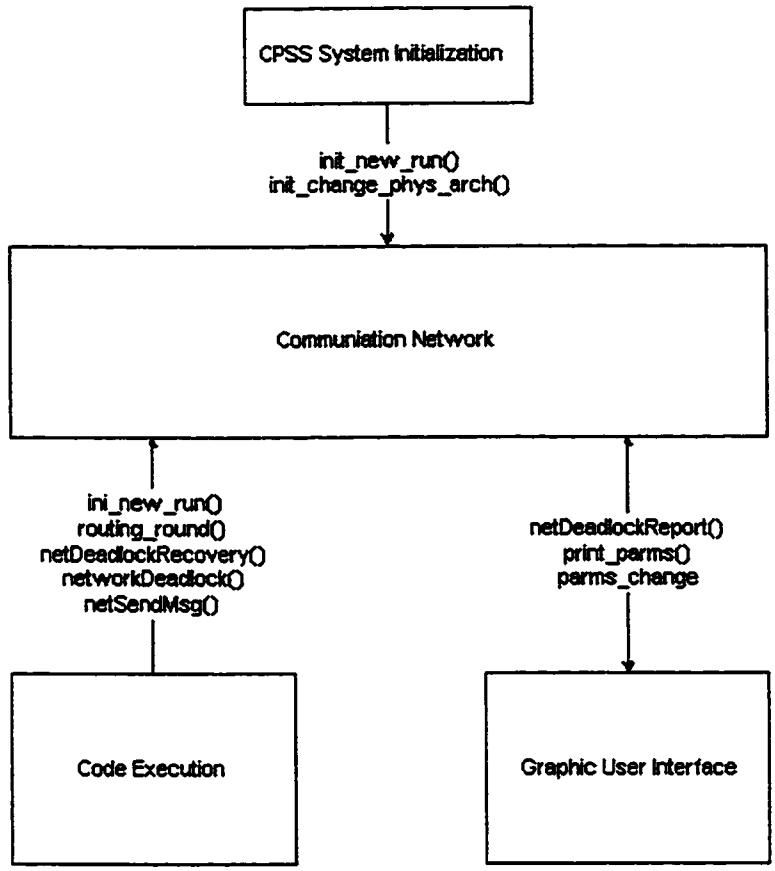

Figure 12. Interaction between Network Simulator and the rest of CPSS

Chapter 5

Implementation of CPSS Network Simulator

The CPSS communication network simulator was described in Chapter 3 and the high level design of CPSS network simulator was discussed in Chapter 4. From the network simulator design objectives, we can see that the object-oriented language can match these design objectives. As the rest parts of CPSS are implemented in C or C++, it is better to adopt the C++ language to implement the communication network simulator. By doing so, the code of the communication network simulator can be integrated into CPSS development environment.

In this chapter, the network implementation details, the related parameters and data structures will be presented. The network parameters are given firstly, as we need the terminology and the knowledge of data structure to describe and understand the network simulator implementation. Then the detailed implementation for each routing technique is presented. Since all routing techniques have common interface functions, the description for the implementation details will follow the interface function prototypes.

5.1 Network Parameters

5.1.1 Network Topology

Typical network topologies are line, ring, mesh, torus and hypercube. In high dimensional networks such as hypercube, the average distance between nodes is small. The routing latency is thus reduced accordingly. However, high dimensional topologies are wire limited. The reason is that the number of physical connections between nodes is limited by the number of available pins and pads on the router and the available chip area for communication related hardware. The more dimensions, the more links in that topology, and thus the more difficult the topology is to fabricate in a limited area.

Low dimensional topologies, such as line, ring, mesh and torus, offer higher wire efficiency. But the average distance between nodes is relatively large. In the implementation of our network simulator, both low and high dimensional topologies are supported.

5.1.2 Message

A message is a logical unit of information for inter-process communication. Messages may have variable lengths. A message usually carries the source and the destination node addresses, the message length, message sent time, the content of the message, and the Process ID which needs to be notified upon the arrival of the message.

The data structure of a message can be represented as below:

source	*dest*	*nbrPackets*	*sendTime*	*next/prev*	*channNum*	*channValIdx*	*processToNotify*

Where:

- *source* is the address of the source node
- *dest* is the address of the destination node
- *nbrPackets* is the message length measured in terms of the number of packets
- *sendTime* is the at which the message will be injected into the network. This time includes all message and packet startup overheads
- *next / prev* are pointers to form a double linked list of messages
- *channNum* is the ID of channel variable
- *channValIdx* is the buffer where the message is deposited for reading
- *processToNotify* is the process to be notified (if any, depends on message type) when the message arrives at the destination

In our CSPP simulator, the message content is stored in channel variable and can be referred by channel number and channel value index. For the description of channel variable, see section 4.1.1.

In packet switch routing technique, a message is divided into a number of fixed-length packets for routing, and in wormhole routed technique, a packet is further divided into a number of fixed-length flits. The packet and flit are discussed in the following section.

5.1.3 Packet

A packet is the basic unit carrying the address of the destination node for routing purposes. Depending on network conditions, packets belonging to a message may arrive

at the destination node out of sequence. Thus a sequence number is required in each packet to allow reassemble of the message.

Typical packet size ranges from 8 to 64 bytes. Factors influencing the choice if packet size include the routing scheme, link bandwidth, router design and network traffic intensity. In our CPSS network simulator, the packet size is a configurable parameter.

In wormhole routed network, every packet is composed if header flits and data flits. Header flit stores the destination address and the sequence number of the packet. The header size thus is depending on the network size and the message length. The buffer size is also affects the header size indirectly.

Packets of a message are routed independently of each other. Therefore they may arrive at the destination out of sequence. When a packet reaches the destination, the *packet* structure is removed for the packets and freed. When all packets of a message arrive the destination, the message is considered to be received completely.

Data needed for routing a packet is stored in a *packet* structure. The structure of *packet* can be represented as following:

msgPtr	*state*	*headNode*	*headLink*	*headLane*	*next / prev*

Where:

- *msgPtr* is a pointer to the *message* structure. Message information need not be depilated in the *packet* structure.

- *state* is the packet state which takes one of the following values: *Init* (just initialized), *InitBlocked* (just initialized and being blocked), *Advancing* (being routed), *Blocked* (being blocked)
- *headNode* is the node where the header flit is currently buffered
- *headLink* is the link where the header flit is currently buffered
- *headLane* is the lane where the header flit is currently buffered
- *next* / *prev* are pointers to form a double linked list of packets

5.1.4 Flit

The concept of flit is introduced when wormhole routed network simulation is considered. A flit is the smallest unit of information transmission in our simulator. Flit size is a configurable parameter in our CPSS network simulator. Affecting factors deciding the choice of flit size include the network size, routing scheme, link bandwidth and router design.

There is no data structure implemented for flit in our simulator. The simulator does not route actual contents of flits but only consider flit IDs at each lane on the path of a packet.

The position of the tail flit is not kept track of explicitly. When the tail flit of a packet leaves a lane buffer, that lane should be deallocated. To implement this, we use field *lastFlitPassed* of the *lane* structure. This field records the ID of the flit which left the lane the most recently. Let the packet size be p, and the flits of a packet be numbered from 1

to p. When field *lastFlitPassed* of a *lane* L reaches value p, this means that the tail flit of the packet occupying L just left the lane. L can be released.

5.1.5 Node

Nodes in our network simulator are identified by absolute IDs. Absolute IDs are computed according to the following rule:

- Line, Ring: assuming that the network has n nodes, the nodes are numbered from 0 to n-1.
- 2D-Mesh and 2D-Torus: assuming that the mesh (torus) has R rows and C columns, the absolute ID of node (r, c) is $r \times C + c$, where $0 \leq r < R$ and $0 \leq c < C$.
- 3D-Mesh and 3D-Torus: assuming that the mesh (torus) has P planes, R rows and C columns, the absolute ID of node (p, r, c) is $(p \times P + r) \times C + c$.
- Hypercube: the absolute ID of a node is the decimal value of the corresponding binary representation of the node address.

The network simulator utilizes absolute IDs so that routing functions are generic and can be used for all types of topology. Only the function computing the next node on the path needs to use Cartesian IDs (for meshes and tori) or binary IDs (for hypercubes).

5.1.6 Virtual Channel / Lane

The concept of virtual channel (lane) is adopted by wormhole routed network simulation. A virtual channel consists of a buffer that can hold one or more flits of a packet and associated state information. Several virtual channels maybe multiplexed on a physical

link and share the link bandwidth. In this subsection, some virtual channel related parameters, concepts and definitions are discussed.

1) Virtual Channel

A virtual channel is a logical link between two adjacent nodes. From the hardware implementation point of view, a virtual channel is composed of

- A flit buffer in the source node which holds flits waiting to use the link
- A physical link between the two nodes, which provides a communication medium between the nodes
- A flit buffer in the receiver node which stores flits just transmitted over the link

Figure 13 shows an example of two virtual channels sharing a physical link. In this diagram, there are two flit buffers in the source node and two flit buffers in the receiver node. One source flit buffer is paired with one receiver flit buffer to form a virtual channel when the link bandwidth is allocated to the pair.

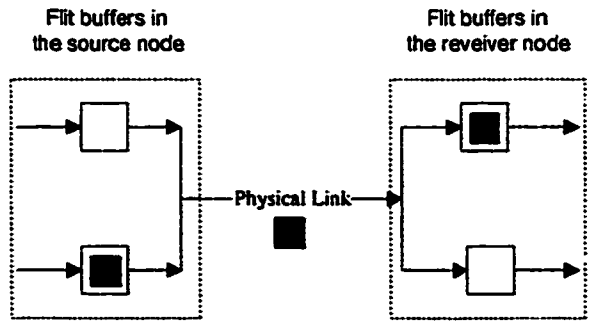

Figure 13. Two virtual channels sharing a physical link

2) Lane

In our CPSS network simulator, such a virtual channel is abstracted into an entity called *lane*. Each lane is associated with a lane buffer which represents a flit buffer at the receiver node. When a flit is deposited into a lane buffer, it is considered to be stored in a flit buffer at the receiver node. This is illustrated in Figure 14.

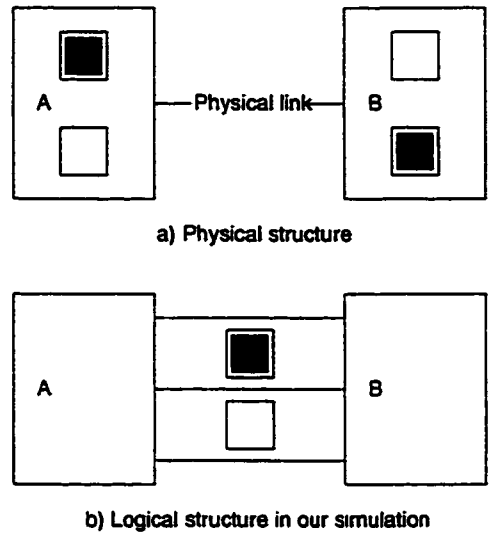

Figure 14. Virtual channel abstraction (Lane)

This abstraction does not improve simulation accuracy, but it helps to speed up simulation time. The term "virtual channel" and "lane" are used alternatively in the following sections.

3) Number of Virtual Channels

In CPSS network simulator, the number of lane per link is a configurable parameter. As we already know, adding more virtual channels help to improve network utilization. However, as the number of virtual channel increase, link schedule becomes more

complicated, requiring additional hardware complexity. In addition, time-sharing of link bandwidth may increase network latency if the traffic is high because the link bandwidth is divided among several messages. Therefore the tradeoff between enhanced network throughput and longer routing latency should be weighted when deciding the number of virtual channels per link.

4) Unidirection ver Bidirection

Virtual channels can be unidirectional or bi-directional. If we defined a pair of opposite unidirectional channels between two adjacent nodes, the implementation and control of this scheme could be quite simply. In this case, the situation, which one channel is very busy while the other is idle, could happen. Thus the physical link is not fully utilized.

Link utilization can be increased by combining two unidirectional lanes into a single bi-directional lane. If the bandwidth of each unidirectional lane is W, then the bandwidth of the corresponding bi-directional lane is $2W$. Link utilization increases at the cost of more complex implementation. In our simulator, we assume that bi-directional virtual channels share a bi-directional physical link.

5) Buffer Size

Each virtual channel is associated with a buffer. The buffer size of a channel is an integral multiple of the flit size. Virtual channel buffers are FIFO queues. Large buffer size may improve network performance, but buffer size equal to packet size will effectively reduce the benefit of wormhole routing to that of packet switching. In our

simulator, buffer size is a configurable parameter. The minimum value is 1 flit, and the maximum value is the packet size in flits.

6) Virtual Channel Allocation Policy

The number of packets sharing a link may be larger than the maximum number of virtual channels multiplexed on that link. When all lanes of the link are busy, incoming messages have to be queued at the corresponding router. Free lanes are then allocated to waiting messages using an allocation policy. FIFO queues are usually used for lane allocation. In real-time networks, packets requesting free lanes are ordered and given free lanes using message priorities. Our simulator uses FIFO queues for lane allocation.

7) Link Bandwidth Allocation Policy

Each physical link is associated with a scheduler that multiplexes data from the virtual channels over the physical link. This allows the virtual channels to time-share the bandwidth of the physical link. A fair scheduling algorithm, such as round-robin, can be used for link band width allocation. In real-time systems, priories of messages occupying the virtual channels can be used for scheduling.

8) Data Structure of Lane

The structure of *lane* can be represented as following:

packetPtr	*state*	*nbrFlits*	*LastFlitPassed*	*prevLink*	*prevLane*

Where:

- *packetPtr* is a pointer to the *packet* structure of the packet currently occupying this lane

- *state* is the lane state which can be *Free*, *Busy* or *FirstLane*. A lane is *busy* if it is currently used by some packets. A lane is *FirstLane* if it is busy and connected to the source node.
- *NbrFlits* is the number of flits currently buffered in this lane
- *LastFlitPassed* is the ID of the flit which left this line the most recently. Assume that the packet size is p and the flits of a packet are numbered from 1 to p. When this field is set to p, we know that the tail flit of the packet just left the lane. Thus we can release the lane
- *prevLink* is the previous link on the path of the packet
- *prevLane* is the lane on the previous link and occupied by this packet

5.1.7 Link

A link is a network pass between two neighbor nodes. Each link can be configured with a number of lanes available on it. During message passing process, a link is searched first to find out the route, then an available lane is checked to verify if there is available lane to forward the message.

Links are numbered based on the topology and absolute IDs of nodes.

- Ring: Assume that the ring network has n nodes numbered from 0 to n - 1. Each node in a ring topology is connected to two links: one is to the left and another is to the right of the node. Since our simulator links are assumed to be bi-directional, the total number of links is n. To make the link numbering scheme easy to understand, we

consider that each node of the ring is "in charge" of the link to its right, and this link has the same ID as the node. Figure 15 a) shows an example with $n = 4$.

- Line: Line topology uses the same link numbering as ring. However the two boundary links connected to nodes 0 and n - 1 are not used. Figure 15 b) shows an example with $n = 4$.
- 2D-Torus: Assuming the torus has R rows and C columns, then the absolute node IDs run from 0 to $R \times C$ - 1. Each node of the torus is connected to four links to the east, west, north, and south side of the node. As links are bi-directional, the total number of links is $2\ R \times C$. We consider that each node k ($0 \le k < RC$) of the torus is "in charge" of two links: link $2k$ to the east side and link $2k + 1$ to the south side of the node. Figure 15 c) shows an example with $R = 3$, $C = 4$.
- 2D-Mesh: 2D-Mesh topology uses the same link numbering as 2D-Torus. However the boundary links are not used. Figure 15 d) shows an example with $R = 3$, $C = 4$.

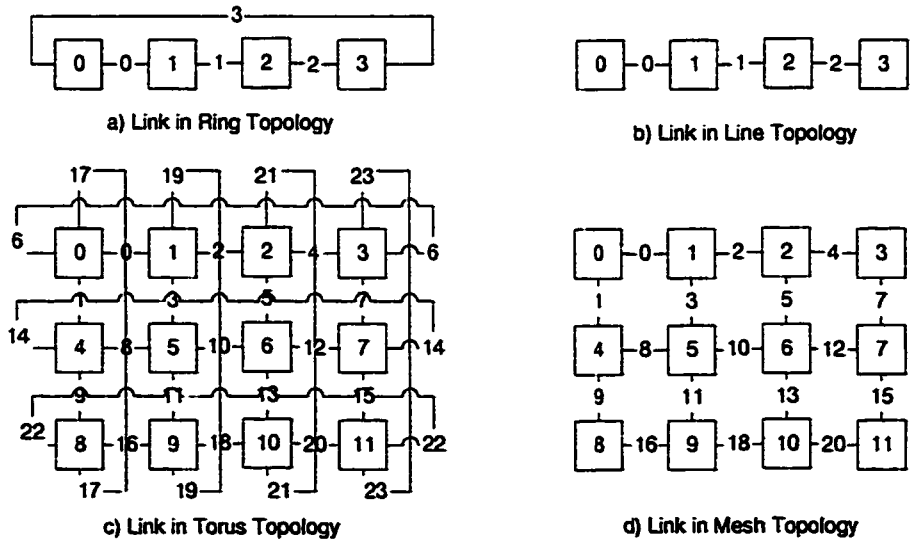

Figure 15. Link Numbering for Two Dimension Topology

- 3D-Torus: Assuming that the 3D-torus has P planets, R rows and C columns, absolute node IDs then run from 0 to $P \times R \times C$ - 1. Each node of the torus is connected to six links to the east, west, north, south, front, and backside of the node. The total number of links is 3 $P \times R \times C$ since the links are bi-directional. We consider that each node k ($0 \leq k < PRC$) of the torus is "in charge" of three links: link $3k$ to the ease, link $3k + 1$ to the south, and link $3k + 2$ to the backside of the node.
- 3D-Mesh: The same link numbering of 3D-torus is applied to 3D-mesh. However the boundary links are not used.
- Hypercube: Let i and j be the absolute IDs of two adjacent nodes. Without loss of generality, assume that $i < j$. Let d be the number of dimensions of the hypercube, and b be the position of the bit where i and j are different (with the rightmost bit being at position 0, $0 \leq b <$ d). The ID for the link connecting node i and j is $i \times d + b$.

The data structure for a link can be represented as following:

Lane	*nbrBusyLanes*	*scheduledLane*	*schedTime*	*queueHead/queueTail*

Where:

- *lane* is an array of dynamic allocated lane structures
- *nbrBusyLanes* is the current number of busy lines
- *scheduledLane* is the lane scheduled to use the link bandwidth in the current network cycle
- *schedTime* is the most recent scheduling time. If this time is equal to the value of the timestamp generator, it means that the link has been scheduled in the current network cycle

- *queueHead/queueTail* are pointers tot he head and tail of the list of packets waiting for free lanes to be allocated. In the current implementation, this queues are FIFO queues

5.2 Implementation for Shortest Path

Shortest Path is the simplest routing technique deployed in our simulation system. It is an ideal case which assumes that there is no network congestion. The basic idea of this technique is to count the number of hops between the source and the destination nodes in order to estimate the message passing time from the source node to the destination node.

The class definition for Shortest Path can be described as following:

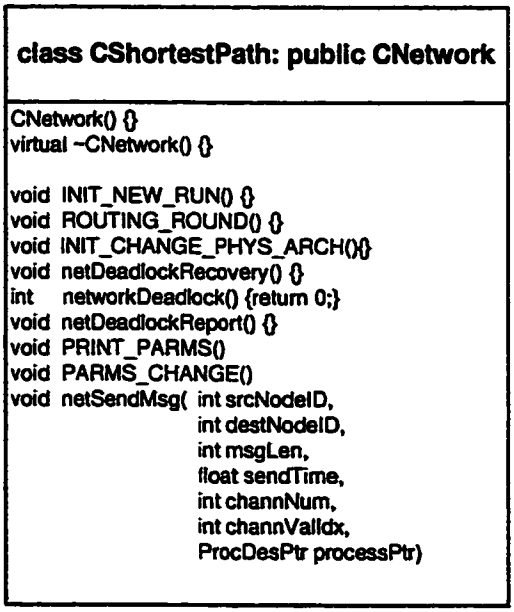

Figure 16. Class CShortestPath Definition

In the subsections below, the implementations for the required interface methods are described.

5.2.1 Constructor and Destructor

From the class CShortestPath definition, we can see that the class constructor CShortestPath()and destructor ~CShortestPath are not implemented. As we discussed before, the class constructor is used to allocate network resources to the specified routing technique, and the class destructor is used to release the resources hold by the specified routing technique. Since the shortest path routing technique does not require any network resources, these methods are not needed.

5.2.2 INIT_NEW_RUN()

The INIT_NEW_RUN() method is to reset the network resources. This method does not release the resources, just reset the related parameters to their initialize values. Since the shortest path routing technique does not need to acquire network resources, this method is not implemented in this case.

5.2.3 ROUTING_ROUND()

The ROUTING_ROUND() method is used when network get its time slice to forward messages. Since the shortest path routing technique does not simulate the message passing procedure, but just estimating the message passing time by simply count the number of hops between the source and the destination nodes. Therefore, this method is not implemented.

5.2.4 INIT_CHANGE_PHYS_ARCH()

The INIT_CHANGE_PHYS_ARCH() method is called whenever the network physical architecture is changed. This method is used to reorganize the routing table for message passing procedure, reallocate lane and link. As the reason we mentioned above, the shortest path routing technique does not actually forward the message, therefore this method is not implemented for shortest path routing technique.

5.2.5 netDeadlockRecovery()

The netDeadlockRecovery() method is called when deadlock happens. This method is used to free entries in request queues of physical links, free list of active packets, free list of active messages and free list of new messages. Since only the wormhole routing technique has the potential to have deadlock happened, in shortest path, this method is not implemented.

5.2.6 networkDeadlock()

The networkDeadlock() is used to detect deadlock. This method is called in jobscheduler(). Whenever a deadlock happens, the simulation will stop and generate report. This method is not implemented in shortest path since there is no possibility to have deadlock happened.

5.2.7 netDeadlockReport()

The netDeadlockReport() is called whenever a deadlock is detected. This method is used to generate report indicating where the deadlock happens. The same reason as mentioned above, this method is not implemented in the shortest path routing technique.

5.2.8 PRINT_PARMS()

In the CPSS simulator, the network parameters are configurable. The PRINT_PARMS() method is used to printout these parameters, like number of lane, link, and packet size, etc. This method is implemented just to display a non-support message in shortest path since shortest path does not dealing with these parameters.

5.2.9 PARMS_CHANGE()

The PARMS_CHANGE() method is used to change the network parameters. As the reason mentioned above, this method is implemented just to display a non-support message in shortest path.

5.2.10 netSendMsg()

In shortest path routing technique, messages are not actually routed through networks. The netSendMsg() method simply calculates the number of hops between the source and the destination nodes according to the network topology to get the network latency directly. The following topologies are considered:

- **Shared:** when the topology is shared, there is no need to pass message. So the elapse time of passing message can be considered as 0

- **FullConnect:** when the topology is full connection, which means there is a connection between each node. So the elapse time of message passing is one hop's jumping time
- **Line:** when the topology is line, which means a message has to pass each node in between to reach the destination. So the total number of node which a message has to pass is |destNode - sourceNode|. Therefore, the elapse time of message passing is |destNode - sourceNode| hop's jumping time
- **Ring:** It is similar to Line except the message can go in the opposite direction to reach the destination, which takes less time
- **Mesh:** From this point on, the message passing time is not only has relationship with number of processors, but it also has relationship with the topology dimension and each dimension size. CPSS has a global variable to hold the physical topology dimension, and its value is assigned when changing the physical architecture type, **void archProc(int topo, int dim, int dimSizes[])**. Messages traveling between distant processors must travel along horizontal or vertical paths between immediate processor. Every pair of processor will have a minimum path-length between them, measured by the sum of the row distance and the column distance
- **Torus:** It is similar to Mesh except the message can go in the opposite direction to reach the destination, which takes less time
- **Hypercube:** In a hypercube interconnection topology, the number of processors is always an exact power of 2. The distance between processors is equal to the number of bit positions in which their processor number is different

5.3 Implementation for Simulated Packet

Simulated Packet is the upgraded version of Shortest Path routing technique. Instead of simulating the message passing procedure directly by counting the number of hops between source and destination nodes, network congestion is considered by introducing the concept of BusyList. Each processor maintains a busy list. In simulated packet, messages are divided into packets and these packets can be delivered at the same time from different processor. When more then one packet arrive the same processor at the same time, it does no consider to be forwarded immediately (This is the point where it is different from Shortest Path). Instead it is pending in the BusyList at that processor.

The class definition for Simulated Packet can be described as following:

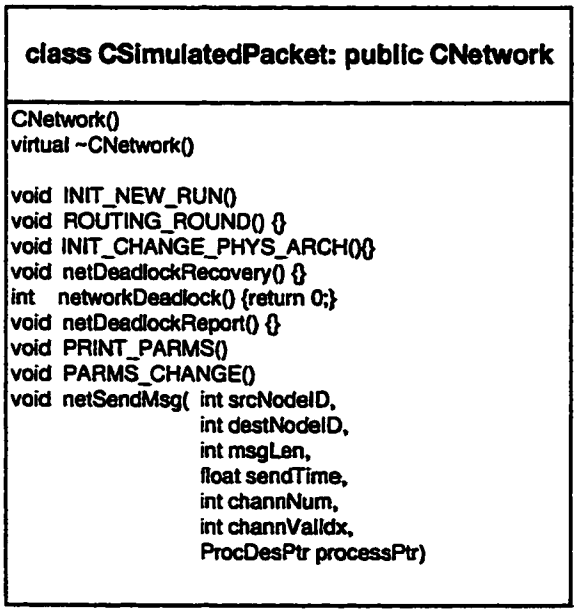

Figure 17. Class CSimulatedPacket Definition

5.3.1 Constructor/Destructor

Since the simulated packet routing technique maintains a BusyList at each processor site. The BusyList is allocated at the class constructor time and freed at the class destructor time.

5.3.2 INIT_NEW_RUN()

This method is called when CPSS restart to run the program. At this time, it is not necessary to release the BusyList, only the messages pending at each processor has to be removed and creates a fresh environment for next run. This method is also called in class destructor to release the list at each processor before release the BusyList in order to avoid memory leak.

The structure of BusyList can be represented as following:

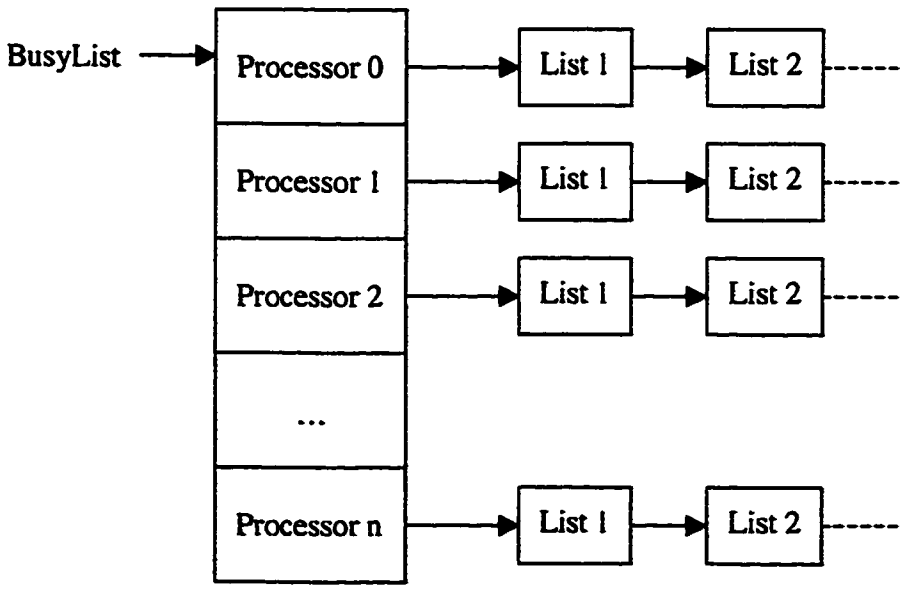

Figure 18. The Structure of BusyList

5.3.3 ROUTING_ROUND()

This method is not implemented since the simulated packet routing technique does not simulate the message passing procedure.

5.3.4 INIT_CHANGE_PHYS_ARCH()

The INIT_CHANGE_PHYS_ARCH() method is called whenever the network physical architecture is changed. This method is used to reorganize the routing table for message passing procedure, reallocate lane and link. As the reason we mentioned above, the simulated packet routing technique does not actually forward the message, therefore this method is not implemented for shortest path routing technique.

5.3.5 netDeadlockRecovery()

The netDeadlockRecovery() method is called when deadlock happens. This method is used to free entries in request queues of physical links, free list of active packets, free list of active messages and free list of new messages. Since only the wormhole routing technique has the potential to have deadlock happened, in simulated packet, this method is not implemented.

5.3.6 networkDeadlock()

The networkDeadlock() is used to detect deadlock. This method is called in jobscheduler(). Whenever a deadlock happens, the simulation will stop and generate

report. This method is not implemented in simulated packet since there is no possibility to have deadlock happened.

5.3.7 netDeadlockReport()

The netDeadlockReport() is called whenever a deadlock is detected. This method is used to generate report indicating where the deadlock happens. The same reason as mentioned above, this method is not implemented in the simulated packet routing technique.

5.3.8 PRINT_PARMS()

The PRINT_PARMS() method is used to printout these parameters, like number of lane, link, and packet size, etc. This method is implemented just to display a non-support message in simulated packet since the simulated packet does not dealing with these parameters.

5.3.9 PARMS_CHANGE()

The PARMS_CHANGE() method is used to change the network parameters. As the reason mentioned above, this method is implemented just to display a non-support message in shortest path.

5.3.10 netSendMsg()

The netSendMsg method in Simulated Packet is similar to the method in Shortest Path. The difference is instead of returning the number of hops between the source and

destination node directly, the network congestion is considered by implementing the BusyList for each processor.

5.4 Implementation for Packet Switch

In packet switch routing technique, the network manager maintains three linked lists. They are new message list *newMsgHead*, active message list *actMsgHead* and packet list *packetHead*. When a message needs to be sent, it is inserted into *newMsgHead* in the order of message send time. When network gets its turn to execute, the new message list is scanned and messages whose send time is no bigger than current time will be removed from *newMsgHead* and appended to *actMsgHead*. In addition, this message will be cut into packets and make it ready to be sent out through network

For packet switching, there is no concept of lane. Each link may have an earliest available time, and each packet has an arrival time to a node. At each round of communication, for each node, find the packet with the smallest arrival time that is not less than the available time of its needed link, send it overt he link, and added packet transmission time to that link's available time as well as the packet's arrival time. If no packet in the node satisfy this condition, then no packets will be forwarded from that node.

The class definition for Packet Switch can be described as following:

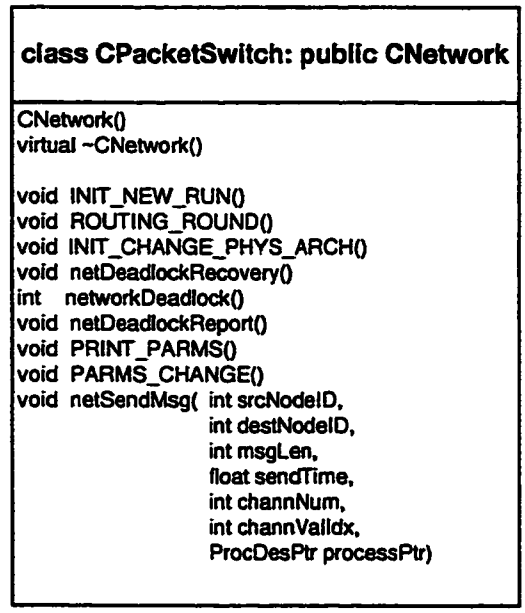

Figure 19. CPacketSwitch Class Definition

5.4.1 Constructor/Destructor

In the constructor of packet switch class, the network resources have to be reserved for the message passing process. In the destructor of packet switch class, the reserved resources have to be released. The resources include the routing table, network links and lanes, and the network related parameters.

5.4.2 INIT_NEW_RUN()

This method is called when CPSS restart to run the program. In packet switch routed network, the parameters which are required to be initialized include the network clock, links, the lane structure on the links, and also pointers to header and tail of new message queue, the active message queue, and packet queue.

5.4.3 ROUTING_ROUND()

The routing_round function is called after all processors get their chance to execute one time slice. The CPSS simulates the parallel programming by assigning each processor a equal time slice. After all processors have taken their time slice, the global clock of the simulator is advanced by one time unit. The routing_round function is executed when the global clock is advanced. The purpose of this function is to scan the message queue and start to route the available message (The available message refers to the message whose send time less than the current time).

5.4.4 INIT_CHANGE_PHYS_ARCH()

The init_change_phys_arch function is called when the system physical architecture has been changed. This function is to free the system resources allocated for the previous physical architecture and reallocate them according to the new physical architecture setting. The system resources mentioned here include routing table, links, lanes and some of the network parameters which dealing with system topology.

5.4.5 netDeadlockRecovery()

The netDeadlockRecovery() method is called when deadlock happens. Since only the wormhole routing technique has the potential to have deadlock happened, in packet switch, this method is not implemented.

5.4.6 networkDeadlock()

The networkDeadlock() is used to detect deadlock. This method is not implemented in packet switch since there is no possibility to have deadlock happened.

5.4.7 netDeadlockReport()

The netDeadlockReport() is called whenever a deadlock is detected. This method is used to generate report indicating where the deadlock happens. The same reason as mentioned above, this method is not implemented in the packet switch routing technique.

5.4.8 PRINT_PARMS()

In packet switch class, the print_params function prints out the configurable parameters for packet switch network. These parameters include number of lanes per link, channel buffer size, packet size, packet header size, packet data size, startup_cost per message, and startup_cost per packet.

5.4.9 PARMS_CHANGE()

This function provides a facility to change the configurable parameters in packet switching routed network. In section 5.1, we have introduced the network parameters and specified which parameters are configurable, here we summarize those which are configurable specially in packet switch. They are number of lanes per link, packet size, channel buffer size, startup_cost per message, startup_cost per packet.

5.4.10 netSendMsg()

The purpose of the netSendMsg function is to forward messages. In packet switching routed network, a message can not be forwarded directly. Whether a message can be forwarded or not, it depends on the availability of network resources. Therefore this function simply insert the message to the new message queue and waiting to be routed when the network get its time slice to execute.

5.5 Implementation for Wormhole Routing

In wormhole routing technique, a communication step has two phases. In the first phase, all packets whose header flits have not reached their destination request the next lanes on their paths. If no free lane is available on a requested link, the requesting packet is queued at the link, waiting for a lane to be released. Otherwise, a free lane is reserved exclusively for this packet.

The second phase is also packet-driven. The network manager attempts to advance unblocked flits of all packets by one link. For each packet p, the network manager visits every lane on the current path of the packet. The visiting order is from the header flit going backward to the tail flit. For each lane L belonging to packet p, if the corresponding link k has not been scheduled in this communication step, then packet p will schedule link k on behalf of the order packets sharing link k. The ID of the lane, which is allowed to use the link in this communication step (if any), is recorded in field *scheduleLand*. If the *scheduleLane* is lane L, the flit in this lane (which belongs to packet p) will be forwarded by one link.

The class definition for Wormhole can be described as following:

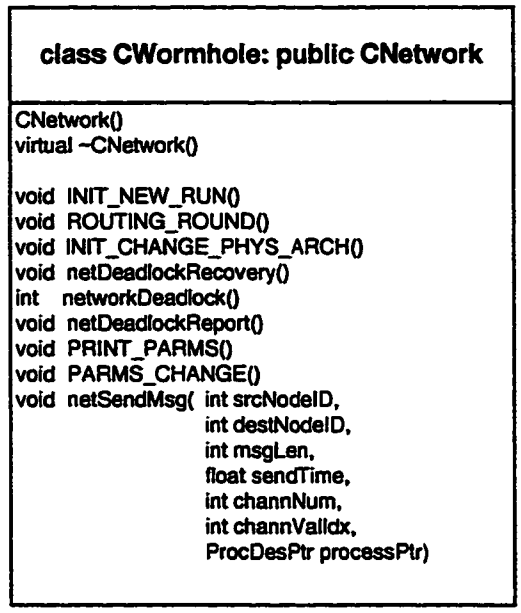

Figure 20. CWormhole Class Definition

5.5.1 Constructor/Destructor

In the constructor of Wormhole class, the network resources have to be reserved for the message passing process. In the destructor of Wormhole class, the reserved resources have to be released. The resources include the routing table, network links and lanes, and the network related parameters.

5.5.2 INIT_NEW_RUN()

This method is called when CPSS restart to run the program. In Wormhole routed network, the parameters which are required to be initialized include the network clock, links, the lane structure on the links, and also pointers to header and tail of new message queue, the active message queue, and packet queue.

5.5.3 ROUTING_ROUND()

The routing_round function is called after all processors get their chance to execute one time slice. The CPSS simulates the parallel programming by assigning each processor a equal time slice. After all processors have taken their time slice, the global clock of the simulator is advanced by one time unit. The routing_round function is executed when the global clock is advanced. The purpose of this function is to scan the message queue and start to route the available message (The available message refers to the message whose send time less than the current time).

5.5.4 INIT_CHANGE_PHYS_ARCH()

The init_change_phys_arch function is called when the system physical architecture has been changed. This function is to free the system resources allocated for the previous physical architecture and reallocate them according to the new physical architecture setting. The system resources mentioned here include routing table, links, lanes and some of the network parameters which dealing with system topology.

5.5.5 netDeadlockRecovery()

Whenever deadlock happens, the simulation system tries to recover it from deadlock by removing the message queues which are waiting for a free link /lane. in addition, all the message/packet lists are released in order to start another new run.

5.5.6 networkDeadlock()

The networkDeadlock is function is to detect the deadlock status. The deadlock flag is set when there is at least one packet which is waiting to be routed but the network system indicates there is no schedulable packet.

5.5.7 netDeadlockReport()

Whenever deadlock is detected, the system will generate a report indicating the current network type and the packet where the deadlock happened.

5.5.8 PRINT_PARMS()

In wormhole routed class, the print_params function prints out the wormhole network configurable parameters. They are number of lanes per link, flit size, channel buffer size, packet size, flit header size, packet data size in flit, startup_cost per message, startup_cost per packet, and the ratio between non_head_flit_speed and head_flit_speed.

5.5.9 PARMS_CHANGE()

This function provides a facility to change the configurable parameters in wormhole routed network. In section 5.1, we have introduced the network parameters and specified which parameters are configurable, here we summarize them all together. The configurable parameters in the wormhole routed network include: number of lanes per link, packet size, flit size, channel buffer size, startup_cost per message, startup_cost per packet, and the ratio between non_head_flit_speed and head_flit_speed.

5.5.10 netSendMsg()

The purpose of the netSendMsg function is to forward messages. In wormhole routed network, a message can not be forwarded directly. Whether a message can be forwarded or not, it depends on the availability of network resources. Therefore this function simply insert the message to the new message queue and waiting to be routed when the network get a chance to be executed.

Chapter 6

Experimental Results

In the previous chapters, the design and implementation of four routing techniques (Shortest Path, Simulated Packet, Packet Switch and Wormhole Routing) are presented. In this chapter, how different routing techniques have influence on the performance statistics of CPSS are illustrated by studying two examples. The purpose of these studies is to testify what we did is correct. One example is a simple parallel program which is running under ring topology and each processor sends a message to its next neighbor. Another example is a more complicated parallel program which is matrix multiplication program. In the following subsection, these two examples are studied individually.

6.1 Example 1: Ring Communication Program

The source code of Ring Communication Program is illustrated in Figure 21. In this program, the virtual architecture is specified as Ring, physical architecture is specified as Line, and each with dimension 8. On each processor, a send buffer z and receive buffer y are defined. The main() function initializes the send buffers for each processor and then invokes f() function to be performed parallel on each processor. The function f() performs message sending and receiving operations.

```
#include "cpc.h"

#define N 8

phyArch line L[N];
arch    ring R[N];
int z[N], y[N];

void main() {
   int i;

   timeOff();
   for (i = 0; i < N; i++)
      z[i] = i;

   timeOn();
   forall (i from 0 to (N-1)) f(i);
}

void f(int i)
{
   barrier(N);
   send(&z[i], 1, 0, i+1);
   receive(&y[i], 1, 0);
}
```

Figure 21. Ring Communication Program

When the CPSS system uses default mapping between physical architecture and virtual architecture, the performance statistics with different routing technique is as following:

Routing Technique	**Sequential Execution time**	**Parallel Execution Time**	**Sequential/ Parallel Execution Time**
Shortest Path	699	330	2.12
Simulated Packet	699	340	2.06
Packet Switch	699	450	1.55
Wormhole Routine	699	490	1.43

Table 1. Performance Statistics for Ring Communication Program

*Note: The execution time could be different when running on different machine, but speedup should be the same.

6.2 Example 2: Matrix Multiplication Program

The source code of Matrix Multiplication program is illustrated in Figure 22. In this program, the virtual architecture is specified as Torus, physical architecture is specified as Mesh, and each with dimension 8 × 8. In order to simplify the computation, matrix multiplication is applied to two 8 × 8 matrix. The main() function first initializes these two 2D matrices with random numbers, then multiplication is applied to these two matrix. Unlike the traditional matrix multiplication, this program takes the advantage of parallel programming by forking the computation to each processor in order to maximum the operation speed.

```
#include "cpc.h"

#define N 8

arch torus T[N][N];
phyArch mesh L[N][N];

channel float Achan[N][N], Bchan[N][N];

void main() {
   float a[N][N], b[N][N], c[N][N];
   int i, j;

   timeOff();
   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         a[i][j] = rand()*5;
   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         b[i][j] = rand()*5;
   timeOn();
   pMultify(a, b, c);
}

void multiply(int row, int col, float myA, float myB, float &mainC) {
   int iter, above, left;
   float myC;

   if (row>0)
      above = row - 1;  // up neighbor
```

```
    else
       above = N - 1;
    if (col>0)
       left  = col - 1;  // left neighbor
    else
       left  = N - 1;
    myC = 0;
    for (iter=0; iter<N; iter++) {
       Achan[row][left] = myA;     // send myA in leftward rotation
       Bchan[above][col] = myB;    // send myB in upward rotation
       myC += myA * myB;
       myA = Achan[row][col];      // receive new myA
       myB = Bchan[row][col];      // receive new myB
    }
    mainC = myC; // Send final value to main process
}

void pMultify(float &a[N][N], float &b[N][N], float &c[N][N]) {
    int i, j;
    forall (i from 0 to N-1)
       forall (j from 0 to N-1)
          fork [i*N+j; Achan[i][j], Bchan[i][j]]
             multiply(i, j, a[i][(j+i) % N], b[(i+j) % N][j], c[i][j]);
}
```

Figure 22. Matrix Multiplication Program

In order to understand this program well, here are some explanations for parallel programming syntax:

- **Forall**: is a parallel form of *for* loop. Each iteration of a *forall* statement creates a child process which will run in parallel with other children processes created by the same *forall.*
- **Fork**: followed by any expression will create a new process which will evaluate the expression and run in parallel with the parent. The parent will continue execution right after the creation of the child without waiting for the child to terminate. The child and the parent will then be running in parallel.

When the CPSS system uses default mapping between physical architecture and virtual architecture, the performance statistics with different routing technique is as following:

Routing Technique	Sequential Execution time	Parallel Execution Time	Sequential/ Parallel Execution Time
Shortest Path	35610	1240	28.72
Simulated Packet	35688	1290	27.67
Packet Switch	35853	3560	10.07
Wormhole Routine	35816	3630	9.87

Table 2. Performance Statistics for Matrix Multiplication Program

*Note: The execution time could be different when running on different machine, but speedup should be the same.

From the above table, we can see that the Shortest Path routing technique takes more time on sequential execution, but less time on parallel execution. Simulated Packet takes little bit more time on parallel execution than the Shortest Path. For the Packet Switch routing technique, the parallel execution time increased significantly. And the Wormhole routing technique takes even more time on parallel execution than the Packet Switch.

Chapter 7

Conclusion and Future Work

This thesis focused on the design and implementation of CPSS communication network using object-oriented methodology. The aim of the design is to provide CPSS an independent, encapsulated communication network module. By extracting the implementation of network module from CPSS, CPSS does not need to aware the changes of communication network, including network topology, routing technique, network parameters.

In the current release of the network module, four types of routing techniques are implemented. They are Shortest Path, Simulated Packet, Packet Switch and Wormhole.

The CPSS communication network module makes the CPEE an excellent environment for developing and fine-turning parallel programs due to the following advantage features:

- Encapsulation: By deploying the object-oriented design, all network related information and implementation details are encapsulated from the other part of CPSS. The advantage is when a person works on the rest of CPSS parts, he/she doesn't need to be interrupted by the communication network code.

- Polymorphism: Our network module supports four types of routing techniques. With this feature, the network module only exposes its abstract interface to the rest of CPSS parts and CPSS does not need to aware which routing technique and what topology the system is running on. This simplifies the development of the other part of CPSS.
- Extensibility: There are four types of routing techniques are supported in our current version of CPSS. Probably there will be a request to support more routing techniques in the future. With the two characteristics listed above, this feature is easy to achieve. Programmers do not need to go through the whole CPSS code to do modifications, and only the network module is need to be accessed. In addition, the developer only need to get familiar with the common network interface, and create a new class which inherence from this abstracted common interface to do the development work.

The following features can be implemented in the future:

- Implementation of more routing techniques. Like we discussed previously, there are virtual cut-through, circuit switch etc. which are not implemented. With the object-oriented design done in this thesis, this work becomes more straightforward.
- Object-oriented design for CPSS. In the current version, only the network module is extracted from CPSS as an independent object. CPSS itself can be design and implemented using object-oriented methodology.

References

[1] BERTSEKAS D. P. and TSITSIKLIS J. N., Parallel and Distributed computation: Numerical Methods, Prentice Hall, 1989

[2] BREWER, E. A., et al, "Proteus: A High Performance Parallel Architecture Simulator", Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, Laboratory of Computer Science, September 1991

[3] CHITTOR S. and ENBODY R., "Performance degradation in large wormhole-routed interprocessor communication networks", Proceedings of the International Conference on Parallel processing, 1990, vol.1, pp.424-428

[4] CHITTOR S. and ENBODY R., "Predicting the effect of mapping on the communication performance of large multicomputers", Proceedings of the International Conference on Parallel Processing, 1991, vol.2, pp.1-4

[5] CORMEN T. H., LEISERSON C. E. and RIVEST Ronald L., Introduction to Algorithms, the MIT Press, 1990

[6] DALLY W. J., "Performance analysis of k-ary n-cube interconnection networks", IEEE Transaction on Computers, June 1990, vol.39, pp.775-785

[7] DALLY W.J., "Virtual-channel flow control", IEEE Transaction on Parallel and Distributed Systems, March 1992, vol.3, no.2, pp.194-205

[8] DAVIS H., GOLDSCHMIDT S. R., and HENNESSY J., "Multiprocessor simulation and tracing using Tango", *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991, vol.2, pp.99-107

[9] DELAGI B. A., et al, "An instrumented architectural simulation system",

Technical Report KSL 86-36, Knowledge System Laboratory, Stanford University, January 1987

[10] DELAGI B. A., et al, "Instrumented architectural simulation", Technical Report KSL 87-65, Knowledge System Laboratory, Stanford University, November 1987

[11] FORD W. H. and TOPP W. R., Introduction to Computing using C++ and Objected Technology, Prentice Hall, 1999

[12] Gao G. et al, "Towards a portable parallel programming environment", *Proceedings of the Supercomputing Symposium*, June 1992, pp.219-228

[13] GOLDSCHMIDT S. and DAVIS H., *Tango Introduction and Tutorial*, Computer System Laboratory, Stanford University, February 1991

[14] HWANG K., *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, 1993

[15] KUMAR V., GRAMA A., GUPTA A. and KARPPIS G., Introduction to Parallel Computing: Design and Analysis of Algorithms, The Benjamin/Cummings Publishing Company, Inc., 1994

[16] LEIGHTON F. T., Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann, 1991

[17] LESTER B. P. and GUTHERIE G. R., "A system for investigating parallel algorithm architecture interaction", *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987, pp.667-670

[18] LESTER, Bruce P., The Art of Parallel Programming, Prentice Hall, 1993

[19] LO V. M., WINDISCH K. and DATTA R., METRICS: A Tool for the Display and Analysis of Mappings in Message-passing Multicomputer. Proceedings of the

Sixth Distributed Memory Computing Conference, April 1991

[20] NI L. M. and MCKINLEY P. K., "A survey of wormhole routing techniques in direct networks", *Computer*, 1993, pp.62-7

[21] OLK E., "PARSE: Simulation of message passing communication networks", *Proceedings of the 27th Annual Simulation Symposium*, 1994, pp.115-124

[22] QUINN M. J., Designing Efficient Algorithms for Parallel Computers. McGraw-Hill, 1987

[23] REIHER E., HUM H. H. J., and SINGH A., "Simulating networks of superscalar processors", *Proceedings of the Supercomputing Symposium*, 1993, pp.125-133

[24] RUMBAUGH J., BLAHA M., PREMERLANI W., EDDY F. and LORENSEN W., Object-Oriented Modeling and Design, Prentice Hall, 1991

[25] SAHA A., "A simulator for real-time parallel processing architectures", *Proceedings of the 28th Annual Simulation Symposium*, 1995, pp.74-83

[26] SEITZ C. L., et al, "The architecture and programming of the Ametek Series 2010 multicomputer", Proceedings of the conference on Hypercube Computers and Concurrent Applications, January 1988, pp.33-36

[27] SHAY W. A., Understanding Data Communications and Networks, PWS Publishing Company, 1995

[28] STROUSTRUP B., The C++ Programming Language, Third Edition, Addison-Wesley, 1997

[29] Paragon XP/S Product Overview, Supercomputer Systems Division, Intel Corporation, Beaverton, OR 97006, 1991

[30] nCUBE 6400 Processor Manual, nCube Company, Beaverton, OR 97006, 1990