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.

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

UMI®

A Visual Performance Debugger for Concordia Parallel Programming Environment

Jing Zhang

A Thesis

in

The Department

of

Computer Science

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

March 2000

ABSTRACT

A Visual Performance Debugger for Concordia Parallel Programming Environment

Jing Zhang

This thesis presents the design and implementation of the Visual Performance Debugger for the Concordia Parallel Programming Environment (CPPE), a simulator for parallel programming environment. The purpose of this visual performance debugger is to provide a rich set of correctness and performance debugging tools for CPPE to make CPPE a real flexible and efficient system for parallel program development.

The challenge in the design of the performance debugger is the comprehensive recognition of the interaction of performance affecting factors and their impact on parallel applications running on a parallel system. We intend to provide the user with the tool set that can gather performance data regarding different performance affecting factors at different level of details to facilitate the performance analysis and fine-tuning. To this end, a comprehensive survey of performance affecting factors for parallel programming is conducted. Based on this recognition, we have designed and implemented a rich set of correctness and performance debugging tools that can significantly help the user in their parallel programming exercise for correctness debugging and performance tune. We provide a visual environment for this performance debugger in order to achieve these goals more easily.

Acknowledgements

I would like to express my cordial gratitude to Dr. Lixin Tao at the end of my study. Since I have been working in companies throughout most of my studying terms, Dr. Tao has given me the most support in my study and the maximum flexibility in the schedule. Almost all my consultation and discussion with Dr. Tao happened after working hours or during the weekends. This thesis will not be completed without Dr. Tao's compassionate support, advice and encouragement.

My thanks also go to the CPPE team: Dr. Lixin Tao, the team leader; Hassan Hosseini and Ai Kong (CPCC), Hoang Uyen Trang Nguyen and Thien Bui (CPSS), which provided the good basis for the contributions of this thesis.

I am grateful to the professors and administrative stuffs in the department of Computer Science, especially Dr. Butler Gregry, who gave the lectures in Software Engineering and Software Design Methodology, Dr. H.F. Li, who taught an excellent course in Computer Architecture and offered me the guidance in graduate study, Mr. Stan, who gave me so much help in the use of computer system and system programming, and Ms. Halina Monkiewicz, her friendliness and administrative support have made my student's life much easier and pleasant.

I also have a very supportive family that backs me up. My wife supported me with her unconditional love, sharing and inspiration throughout my study. I am also grateful to my parents for their encouragement and help.

Table of Contents

List of Figures ..... **x**

List of Tables ..... **xii**

1 Introduction **1**

1.1 Motivations 2

1.2 General Structure of CPPE 4

1.3 Contributions of This Thesis 6

1.3.1 Correctness Debugging Tools 6

1.3.2 Performance Debugging Tools 6

1.3.3 Graphical User Interface 7

1.4 Thesis Outline 7

2 Literature Survey **9**

2.1 Sources of Performance Degradation 10

2.1.1 Performance Affecting Factors in Communication 11

2.1.2 Performance Affecting Factors in Parallel Computing 14

2.1.3 Parallel Algorithms and Performance 16

2.1.4 Summary 17

2.2 Parallel Computer Simulators and Performance Debugging 17

2.2.1 Direct Execution Simulation System 17

2.2.2 Direct Execution with Code Augmentation 19

2.2.3 Functional Simulation System 25

2.2.4 CPSS Simulation Technique 28

2.3 Graphic User Interface 29

3 The CPSS System Architecture and Debugging Environment.....31

3.1 Design Objectives 31

3.2 CPSS High-Level Design and Debugging Environment 33

3.2.1 The General Structure of the CPSS 33

3.2.2 The Code Execution Module 36

3.2.3 The Network Module 38

3.3 High-Level Design of Debugging Tools 39

3.3.1 Correctness Debugging 39

3.3.2 Performance Debugging 42

3.3.3 Graphical User Interface 45

4 The Design of the Debugging Monitor 46

4.1 Correctness Debugging Tools 46

4.1.1 Setting and Clearing Source Breakpoints 47

4.1.2 Stepping Through a Process 51

4.1.3 Viewing Source Code and vCode 54

4.1.4 Tracing Variables 56

4.1.5 Examining Status of Process 76

4.1.6 Examining Memory Contents 79

4.2 Performance Debugging Tools 80

4.2.1 Setting Time Breakpoint: Alarm 81

4.2.2 Setting Network Architectures 83

4.2.3 Vary Processor Speed 89
4.2.4 Program Mapping 90
4.2.5 Performance Statistics 102
5 Graphical User Interface 111
5.1 Design Objectives 111
5.2 Approaches to Realize the Design Objectives 112
5.2.1 Unified CPPE Development Environment 112
5.2.2 User Friendliness 114
5.2.3 Easy Configuration and Multi-Platform Support 115
5.3 GUI for UNIX Platform 115
5.3.1 Main Function 115
5.3.2 Main Frame 117
5.3.3 Configuration 119
5.3.4 Data Structures 119
5.4 GUI for Windows Platform 122
5.4.1 Main Frame 122
5.4.2 Configuration 123
5.4.3 Data Structures 124
6 Example Applications of Debugging Tools 128
6.1 Virtual-to-physical Mapping And Performance 131
6.2 Topology And Performance 132
6.3 Communication Parameters And Performance 132
7 Conclusion and Future Work 134

Appendix A CPPE User's Manual For UNIX Platform.....137

1 Configuration 137

1.1 Environment Files 137

1.2 Environment Variables 138

1.3 Start The CPPE Program 139

1.4 Configuration Setup 139

2 CPPE Functionality 140

2.1 Create Application Program Source File 141

2.2 Compile Application Program Source File 141

2.3 Execution And Debugging 143

2.4 Network Architectures and Mapping 147

2.5 Program Performance Statistics 154

Appendix B CPPE User's Manual For Windows Platform 158

1 Configuration 158

1.1 Configuration File 158

1.2 Environment variables 158

1.3 Configuration Setup 159

2 CPPE Functionality 160

2.1 Create Application Program Source File 161

2.2 Compile Application Program Source File 161

2.3 Execution And Debugging 162

2.4 Network Architecture And Mapping 167

2.5 Program Performance Statistics 173

Bibliography.....177

List of Figures

Figure 1: General structure of the CPPE 5

Figure 2: Model of the multicomputer architecture 9

Figure 3 : CPSS structure and operations 35

Figure 4 : Data structure of Breakpoint table in C 50

Figure 5 : Memory management of process stacks 58

Figure 6 : Data structures of memory pool in C 60

Figure 7 : The process of retrieving the value of a variable 61

Figure 8 : The process of retrieving the value of a structure variable 66

Figure 9 : The process of retrieving the value of an array variable 69

Figure 10 : Data structure of Channel variable in C 72

Figure 11 : Retrieve messages from a Channel variable 73

Figure 12 : Data structures in C for Trace function 75

Figure 13 : Data structure of Process Control Block in C 78

Figure 14 : Mapping of a process to a virtual processor then to a physical processor 92

Figure 15 : Algorithm for Identity mapping in C 94

Figure 16 : Algorithm for Random mapping in C 95

Figure 17 : Algorithm for Default mapping in C 96

Figure 18 : Algorithm for Ring-to-Line Mapping in C 97

Figure 19 : Mesh and Torus topology 98

Figure 20 :Algorithm for Torus-to-Mesh mapping in C 100

Figure 21 : Data structure related to Physical Processor Descriptor 107

Figure 22 : Performance profile of a parallel program 108

Figure 23 : Event handling of a MOTIF application 117

Figure 24 : CPPE GUI on UNIX workstation platform 118

Figure 25 : Application widget hierarchy in CPPE GUI 120

Figure 26 : Main frame of CPPE on Windows platform 123

Figure 27 : Major classes and their relationship in CPPE GUI 124

Figure 28 : Matrix multiplication on an 8x8 torus 131

List of Tables

Table 1: vCode's that have variable reference 76

Table 2: Process-to-physical-processor mapping 93

Table 3 : Ring-to-Line mapping 97

Table 4: Performance statistics for virtual-to-physical-architecture mapping 131

Table 5: Performance statistics for topologies 132

Table 6: Performance statistics for packet size 133

Chapter 1

Introduction

Computing power has always been an issue in research and industry. Despite the tremendous growth of processor speed over years, there is still a wide range of important computational problems in science and engineering that require much greater computer speed. Parallel programming has shown its potential to meet the demands for high computing power. However, parallel programming is difficult and error-prone, not only because parallel processes are difficult to trace and debug, but also due to the fact that the performance of parallel program is influenced by many hardware and software factors, such as algorithm design, system architecture, routing technique, and networking speed. To make matter even complicated, some algorithms may be more suitable for a particular architecture than others. The performance of a particular parallel program often has to be evaluated through trial and error.

In order to evaluate and improve the performance of parallel application effectively, simulation techniques have been studied in the hope to provide the necessary environment and software tools. CPPE (Concordia Parallel Programming Environment) is one of the simulated parallel systems being studied and developed in our group, aimed to provide users with flexible and efficient software tools for developing parallel programs, evaluating and optimizing performance of parallel applications.

As part of the CPPE project, the objective of the research associated with this thesis is to study the performance affecting factors in parallel computing, design and implement a visual performance debugger for CPPE. The visual performance debugger will provide a rich set of correctness and performance debugging tools to make CPPE a flexible and efficient system for parallel program development.

In this chapter, we first discuss the motivations and objectives of this research. Then we present the contributions of this thesis. At the end we give a thesis outline.

1.1 Motivations

Parallel computers can be classified into two categories: shared-memory multi-processor and message-passing multicomputer. With the advancement of hardware technology, high-speed networks and efficient routing techniques have made message-passing multicomputer the developing trend for parallel computing, because it is more scalable due to the distributed nature of local memories.

On a parallel computer architecture, there are many factors that can affect the performance of a parallel program. Users often have to write a parallel program and run it on an actual parallel system to verify the correctness and observe the performance of the program through trial and error. However, The trial and error approach on a real machine is inefficient because of the following reasons:

1. Real parallel computers are expensive resources that are available to only a restricted number of users. Testing and debugging tools supported on real parallel computer are currently very limited.

2. Real parallel computers are non-deterministic in nature in terms of the network performance and the probability of error. The probability for some bugs to occur may be one over ten thousand. It is vary hard to test, debug and tune a parallel program on a real parallel computer machine.

3. The performance of a parallel program is affected by system size and system topology. The performance of an application may be good on small-sized systems but degrade tremendously as system size increases. An algorithm may be efficient on a particular system topology but perform badly on another topology. To achieve high performance, the designer should study the scalability and communication complexity of parallel algorithms. However, topologies and sizes supported by a real multi-computer are restricted within small ranges. This limitation does not allow for studies of scalability and communication complexity of parallel algorithms.

Because of the limitations of the real machines, many studies of parallel algorithms have been conducted on an analytical basis – the analytical modeling. Performance aspects of a parallel program under specific conditions may be estimated using mathematical formulation. However, this approach is suitable only for simple applications and small computer systems. Parallel computer systems and their applications are sufficiently complex to make analytical modeling very difficult and inaccurate.

Because of the difficulty of using real parallel computers and analytical modeling, the simulation approach has been studied by several research groups. CPPE is one of the projects of that kind. CPPE provides a parallel system simulator called CPSS (Concordia

Parallel System Simulator) which runs sequential software on a uniprocessor to emulate program execution on a real parallel computer. Its objective is to provide a parallel programming environment that allows users to study impacts of system and software factors on program performance and locate performance bottlenecks in the program.

The simulator CPSS is supposed to accurately mimic the behavior of the real parallel machine and yield correct program outputs as if the program has been executed on the real machine. In addition, it should provide users with adequate and useful simulator tools and models for evaluating parallel architectures and the performance of parallel programs on these architectures. Users should be able to use these tools to observe the effects of all the affecting factors on their application so as to detect system bottleneck and thus optimize performance of the applications, which is referred to as *performance debugging*.

The objective of this thesis is to give a comprehensive review of performance affecting factors of parallel computing, design and implement a visual performance debugger for CPPE, which allows users to study the impacts of system architectures and software algorithms on the program performance.

1.2 General Structure of CPPE

This thesis is part of the research project Concordia Parallel Programming Environment (CPPE). CPPE consists of two major modules (Figure 1):

1. Concordia Parallel C Compiler (CPCC): The CPCC accepts parallel programs written in the CPC (Concordia Parallel C) language and generates virtual machine code (vCode) which will be the input to the CPSS.

2. Concordia Parallel System Simulator (CPSS): The CPSS reads in the intermediate code produced by the CPCC, simulates execution of the application, yields programs outputs.

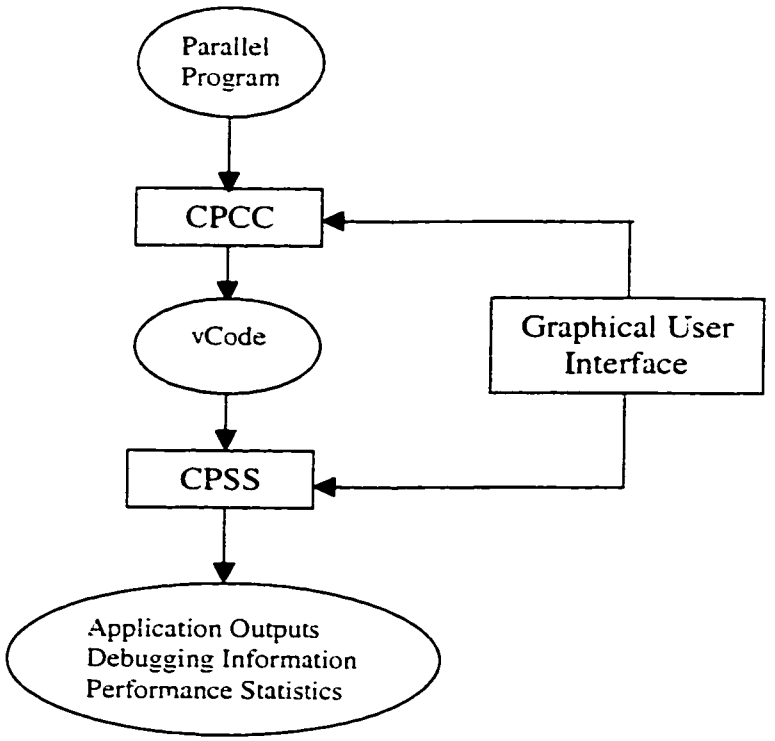


Figure 1: General structure of the CPPE

As a major component of the CPPE, CPSS in turn is made of three main components:

1. Code Execution Module (CEM): The CEM is the processing element of a simulated parallel system. It executes the vCode produced by the CPCC.

2. Network Module: The roll of the network module is to allocate network resources to messages, route and deliver messages, and detect deadlock in the network.

3. Performance Debugger: The performance debugger provides a set of software tools to facilitate program testing and debugging.

The scope of this thesis is to design and implement the Performance Debugger of CPSS and provide a graphic user interface (GUI) for the CPPE. Together we call the software Visual Performance Debugger.

1.3 Contributions of This Thesis

The visual performance debugger includes three major components: correctness debugging tools, performance debugging tools and graphical user interface.

1.3.1 Correctness Debugging Tools

The design concepts of correctness debugging tools are borrowed from sequential programming environment [27]. We concentrate on the most useful debugging tools existing in other sequential debuggers, plus the specific debugging tools necessary for parallel programming environment.

1.3.2 Performance Debugging Tools

The performance debugging tools are meant to provide the performance data and statistics for parallel execution at various levels of details in order to help the user locate performance bottleneck and fine-tune the program. The design of performance debugging

tools is based on the comprehensive study of performance affecting factors of parallel computing. This study enables us to define the most informative and useful performance data and statistics so that we can design corresponding tools to provide such information.

One of the distinguish characteristics of our performance debugger is the ability to support virtual architecture programming and run-time mapping. The user writes an application using the virtual architecture most natural to the application. At run-time the virtual architecture will be mapped to the available physical architecture based on the user-specified virtual-to-physical mapping pattern. For programs with multiple parallel phases, the user can specify a mapping pattern at the beginning of each parallel phase.

1.3.3 Graphical User Interface

The visual performance debugger has a graphic user interface (GUI) which provides a user-friendly developing environment, making the use of debugging tools and the analysis of execution results easy. We provide graphic user interface for versions running on both UNIX and Windows platforms.

The efficiency of our visual performance debugger is tested by actually running some parallel application programs and then analyzing the performance data and statistics gathered from the performance debugging tools. Our simulation results show a good match with the theoretical analysis of the performance under different software and network environment.

1.4 Thesis Outline

In chapter 2, we give a comprehensive review of the performance affecting

factors of parallel computing in both parallel computing and network communication. Chapter 3 gives an overview of CPSS and debugging environment in CPSS, identifies and rationalizes the design of debugging tools in CPSS. Chapter 4 details the design of correctness and performance debugging tools. Chapter 5 describes the design and data structures of graphic user interface. Chapter 6 are some actual parallel program execution and performance statistics using the performance debugging tools built in this thesis. Chapter 7 provides a summary of the thesis and suggestions for future work. Appendix A provides a user manual of CPPE for UNIX platform. Appendix B provides a user manual of CPPE for Windows platform

Chapter 2

Literature Survey

The general structure of a multicomputer system can be illustrated in Figure 2. Each processor has its own local memory and processing unit so that it can compute in a self-sufficient manner, using the data stored in its own local memory. Each processor also has a function unit called router so that every processor can send and receive data from any other processor, using the message-passing communication network. The advantage of multicomputer structure over the multiprocessor include better utilization of memory access locality and support of scalable parallel systems.

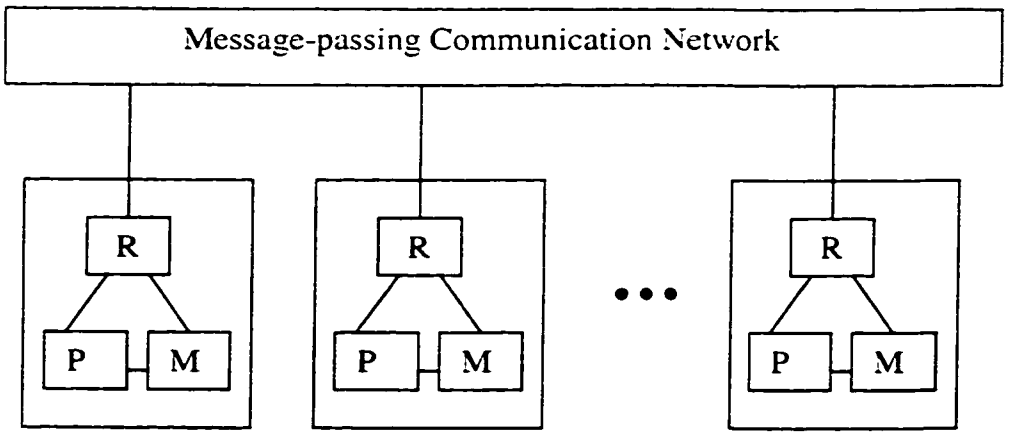


Figure 2: Model of the multicomputer architecture

However, the advantage of multicomputer does not come without cost. The cost is the data communication between different processors. There is another type of cost

concerning the use of the parallel computer system. From a parallel program designer's perspective, parallel computer system makes the parallel programs more difficult to develop, debug and fine-tune the performance. Therefore, a set of user-friendly debugging tools are indispensable in a parallel development system.

CPSS is a simulator of parallel development system which is intended to provide accurate simulation of both computation and communication activities, at the same time, with a flexible and user-friendly environment for correctness and performance debugging of parallel program development. In order to identify and design the debugging tools which are helpful to the users, we should first understand the computation and communication behavior in a parallel system and identify the performance affecting factors in the computation and communication process.

In this chapter, we first present a literature review on performance affecting factors of parallel applications in the perspectives of both parallel computing and communication. Then we present a review of existing code execution simulation techniques and their debugging functionality.

2.1 Sources of Performance Degradation

In order to design effective performance debugging tools for monitoring program performance, first we need to find out the major practical sources of performance degradation of parallel programs when running on real multiprocessors or multicomputers. Some performance affecting factors are typical for multiprocessors, some are typical for multicomputers, while some factors exist for both multiprocessors and multicomputers. The performance of parallel programs are affected by the

combination of software and hardware characteristics. Therefore, without effective performance monitoring tools, it would be hard to predict the final performance output.

2.1.1 Performance Affecting Factors in Communication

In a multicomputer, each processor has its own private physical memory for storing and retrieving data during computation. Each processor also has one or more direct connections to other processors, through which data can be transmitted. If processors have no direct connection, they will communicate through intermediate processors that forward the data. The overall pattern of the direct connections between processors is called the multicomputer topology.

Topology and Performance

There is a wide variety of different types of communication topologies that can be used in parallel systems. The goal of these topologies is to try to reduce the cost and complexity of the network, while rapid communication between processors is achieved. Ideally, the communication cost will be reduced to minimum if there is a direct connection between each pair of processors. However, it is generally not possible in a large multicomputer system to provide a connection between every pair of processors, because this would need n^2 number of connections for n processors. This is not only too expensive, but also practically unachievable for a large multicomputer system. Also, there is a trade-off between cost and performance. In order to maintain a reasonable cost/performance rate and allow the system to be easily scaled up to a large number of processors, different topologies have been developed for multicomputers.

There are two important parameters that characterize each topology: the connectivity, which is the number of direct connections per processor, and the diameter, which is the maximum number of intermediate connections required for distant processors to communicate. The connectivity of a topology is an important affecting factor of network cost, while the diameter is an important affecting factor in the performance of the network. Simpler topologies usually have lower connectivity and therefore a higher diameter. The more complex topologies usually have higher connectivity and therefore a lower diameter. The fact is, we can not make a simple association between performance and topology. The performance is usually affected by a combination of topology and other factors such as parallel algorithm [21][22], message routing technique [15], network speed, data and program mapping [24], and operating system [25]. This fact justifies the existence of different topologies and the requirement for efficient performance monitoring tools.

The multicomputer topology is determined by its underlying interconnection network. CPSS is able to simulate all the widely used network topologies including line, ring, mesh, torus, hypercube and full-connected [23]. Through the performance debugger, user can specify different topologies before simulation without recompile of the simulator software and the user application program. This provides a flexible performance debugging environment.

Communication Delay

The overhead caused by communication delay is typical in multicomputers where the processors interact with each other through message-passing communication. In a

multicomputer, each processor has a separate local memory module for storing its data and program code. In order for all the processors to work together on a single computational problem they must communicate and exchange data during execution.

Processor communication is accomplished by message-passing via processor-to-processor communication link. The basic unit of communication from a programmer's point of view is a *message*, which consists of many bits of data. There are time delays by each message transmission because of the following reasons [26]:

- Transmission time: The communication link has a certain maximum bandwidth. The transmission time is significant compared with the processor speed. In addition to the actual physical transmission of the data bits, the communication module also needs to perform other functions to ensure that data is sent and received correctly. If error does occur, the message may need to be retransmitted.

- Processing time: The communication module needs to perform some computation for every message. The computation includes error detection using the checksum, routing decision based on the source and destination processor IDs.

- Waiting time: The waiting time results from the delay due to congestion in the communication network. Some algorithm may incur considerable amount of message passing, causing heavy network traffic so that messages build up and are delayed.

The actual cost of message transmission can also be affected by the network topology and communication parameters such as packet size, flit size, number of virtual channels per link, and buffer size of the virtual channel [13][14][35][36]. Significant

efforts have been made in both network design and program algorithm design. The ultimate goal is to minimizing the frequency of communication and the distance traveled by each message, and to minimize the communication cost under specific network architecture. However, again because of the complexity of parallel programming, it is hard to accomplish the goal without efficient performance monitoring tools.

The performance debugger developed for CPPE enables user to configure network architecture by redefining the communication parameters and costs without recompile of the simulator software and application programs. Performance statistics enables user to easily obtain the program performance under different network architectures.

2.1.2 Performance Affecting Factors in Parallel Computing

Memory Contention And synchronization Delay

Memory contention is typically a problem when there is shared memory such as in multiprocessors. Processor execution is delayed while waiting to gain access to the shared memory that is currently being used by another processor. When global variables are shared by a large number of processors involved in a parallel computing, memory contention can cause significant performance degradation.

When parallel processes synchronize, one processor may be forced to wait until certain condition is satisfied. So some processes may be idle for a considerable amount of time that causes performance bottlenecks and a reduction of overall speedup.

Because of the memory contention and synchronization delay, the overall performance of parallel computing could be affected to a variety of extents. Users need to

obtain the overall performance data and statistics in order to fine-tune the parallel algorithms.

Excessive Sequential Execution and Process Creation Cost

In any parallel program, there are always portions that are purely sequential code. Certain types of centralized operations, such as initializations, are executed sequentially in one processor before any parallel processes are created. In some algorithm, the sequential code may significantly limit the overall speedup that can be achieved from parallel execution.

The creation of parallel processes requires a certain amount of execution time. If the created processes are relatively short in duration, the process creation overhead may not be compensated by the savings of time due to parallelism. That means, parallel program may not necessarily speed up the execution in some algorithms.

Both scenarios hide the efficiency of parallel computing. In order to realize the actual gain from a certain pattern of parallel computing, we need to provide the performance data for any fragment of program execution in addition to the overall performance monitoring. Typically user may want to obtain speedup data for a pure parallel computing fragment without the counting of cost for sequential computing fragment and parallel process creation.

Load Imbalance

In some parallel programs, computing tasks are generated dynamically in an

unpredictable manner and must be assigned to processors as they are generated. The result is the possibility that some processors may be idle while others have more computing load than they can handle.

Load imbalance is most likely caused by the difference between the optimal computing architecture for a particular algorithm and the available physical parallel architecture. In order to evaluate the efficiency of a particular parallel algorithm on different physical architectures, we need to provide the user with the ability to map a virtual architecture which is optimal to the parallel algorithm to different physical architectures and study the performance under these virtual-to-physical mapping styles.

2.1.3 Parallel Algorithms and Performance

There has been a considerable amount of research in the field of parallel algorithm design for a wide range of practical problems [28, 29, 30]. The most common parallel algorithms include data parallelism, synchronous iteration, replicated workers and pipeline computation. For parallel application programmers, such research help them to fully utilize the enormous computing potential of parallel computers. For us, it helps us to be aware of the different parallel algorithms and their pros and cons for different network architectures, so we can design meaningful test cases to evaluate our system design and demonstrate the usage of our performance debugging tools. By comparing the actual performance output gathered from the performance debugger with the anticipated performance based on the analysis of parallel algorithms and network architecture, we can not only evaluate the efficiency of the performance debugging tools, but also evaluate the correctness and efficiency of the CPSS network simulation system.

2.1.4 Summary

In this section, we summarized the major performance affecting factors in parallel programming. There is no simple formula to calculate the relative weight of each factor contributing to the overall performance, because the performance of parallel programs are affected by a combination of many software and hardware characteristics of the multiprocessors or multicomputers. Effective performance monitoring tools are required for performance debugging.

2.2 Parallel Computer Simulators and Performance Debugging

Several simulation systems for parallel computers have been developed [1][2][3][4][5]. Existing simulation techniques can be classified into three categories: direct execution, direct execution with code augmentation and functional simulation. In this session, we provide a review on these three types of simulation systems concerning the relationship between the simulation mechanism and their debugging functionality.

2.2.1 Direct Execution Simulation System

In direct execution simulation systems, a parallel program is first compiled into object code which is in the assembly language of the host computer. During compile, the compiler identifies two kinds of instructions for the purpose of simulation: local instructions and non-local instructions. An instruction is local if it has effects only on the

local processor, such as accessing variables residing in the local memory. Non-local instructions such as sending messages to a remote processor, in contrast, impact another part of the system such as a remote processor. In the process of simulation, local instructions will be executed directly by host processes and timed with the host's machine clock, while non-local instructions will be simulated via a procedure call which interprets the instruction at the functional level.

Direct execution technique is generally faster compared with the functional simulation technique, because local instructions are executed directly instead of being interpreted. However, it suffers from a major drawback: difficult debugging in terms of feasibility and accuracy.

In terms of feasibility, local instructions are directly executed by the host and the simulating engine does not have much control on the execution of local instructions. Thus it is very difficult to establish the connection between user application code and the low-level simulation activities. Such connection is essential for in-session debugging and fine-tuning an application. (In-session debugging refers to the debugging interaction between users and the program during execution of the program. Examples of in-session debugging are stepping through the program, setting breakpoint, and variable tracing after breakpoints).

In terms of accuracy, direct execution suffers an even more drawback of low accuracy because of the following reasons:

- Because the simulation is timed with the host's clock, which is usually the workstation clock with coarse granularity. Therefore, the timing is not accurate because 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.

- With direct execution technique, monitoring code is often needed because the difficulty of in-session debugging. Such code fragments would not be executed on the target machine. Because there is no way to distinguish monitoring code from the application code, the direct execution technique also times the monitoring code and this will affect the overall execution time, making the simulation even less accurate.

Because of the difficulty of debugging, debugging tools provided by direct-execution simulators are very limited and based primarily on monitoring code added to the application and the simulation engine. In general, direct-execution simulator is not suitable to accurately simulate both computation and communication activities of a target parallel computer and to provide a user-friendly debugging environment.

An example simulator using direct execution technique is the CARE simulator [6][7] which simulates the execution of LISP code.

2.2.2 Direct Execution with Code Augmentation

This approach enhances the pure direct execution technique by adding cycle counts of local instructions to the object code during the compilation phase. The cycle counts of an instruction is the time the target system would take to execute that instruction. Cycle counts 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 accumulated using cycle counts added to the object code. This results in a more accurate simulation than the pure direct execution technique.

Like the pure direct execution technique, code-augmented direct execution is generally faster than functional simulation since local instructions are executed directly rather than being interpreted. On the other hand, code-augmented direct execution offers more accuracy to simulation results than pure direct execution.

However, the problem of difficult debugging still exists. In fact, correctness and performance debugging in direct-execution simulators (both pure and code-augmented) relies heavily on the *instrumented software* technique due to the difficulty of in-session debugging. In the instrumented software approach, additional code is inserted into the simulation engine and the application to monitor the simulation. Adding monitoring code to the simulation engine does not cause any side-effect except that the added code may slow down the simulation. However, adding monitoring code to cycle-counted code (i.e. local instruction blocks) can be problematic. A simple addition will change the behavior of the simulation since the monitoring code is also included in the cycle counting. Conditional compilation flags or macros can be used to exclude the cost of the added monitoring code [2]. However, even with conditional compilation flags or macros, the addition may change the behavior of the application. This is because the additional code may affect the surrounding code indirectly. For example, if the additional code uses several registers, the surrounding code may spill more registers than the previous version (which contains no monitor code). This would increase the cost and thus could 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 Ptoteus [2], Tango [3][8], EPPP [4][9], and PARSE [5].

Proteus

Proteus [2] is developed at MIT in 1991, using 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. A code-augmenting program will then add cycle counts to local instructions of the object code. The compiled code is first 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 of a basic block is then matched with a cycle count by looking up a table. The cycle counts of all the instructions in that basic block are then summed and an instruction updating a global cycle counter is added at the end of the block. The cost of each basic block is thus a fixed number and determined at compile time.

Proteus' debugging capability 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 regarding its debugging functionality:

- Although the simulation results are improved compared with pure direct execution because of the use of cycle counts, the timing results may still not be accurate because the cost of each 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 operands (or cache hits if the target machine is a shared-memory architecture).

- The simulator can simulate accurately only a limited set of architectures whose instruction set is similar to that of the host. If the instruction set of the target machine is quite different from that of the host, the assignment of a cycle count to a local assembly instruction is no longer accurate.

- 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 then needs to be re-compiled and linked with the user application. This is not convenient, for example, for experimenting with different network topologies or program mapping. The experiment would require to run the same program on different architectures of varied sizes. The simulator must be modified, re-compiled and linked with the application code every time the topology or system size is changed.

- Debugging capability 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 [3][8]. Tango and Proteus are quite similar, however, Tango simulates only shared-memory architectures. It was

implemented for studying shared-memory behaviors, shared-memory synchronization and concurrency abstractions and for architectural evaluation [3].

Like Proteus, Tango may produce inaccurate simulation results due to fixed costs of local instruction blocks calculated at compile time. Tango does not support any in-session debugging tools. Debugging and statistics data are provided using the instrumented software approach. Many kinds of trace file are generated. System events are recorded in trace files. Program outputs are logged in an output file. There are also a process summary file and an event trace file. This is not a user-friendly debugging environment for parallel applications. Debugging tools are not adequately provided for code development or performance fine-tuning.

EPPP Project

The EPPP (Environment for Portable Parallel Programming) simulator [4][9] is in fact an extended version of Proteus. In this simulator, the augmentation phase is enhanced to accurately simulate a particular target architecture whose instruction set differs from that of the host. An application program will first be compiled and optimized as it would be on the target system. The intermediate code just produced will then be augmented with cycle counts. A second pass on the augmented intermediate code will generate assembly code for execution on the host.

The above enhancement requires the compiler to be modified specifically for each different target architecture. This is a major task calling for much time and effort. Therefore, the target architectures of the EPPP simulator have been so far limited to only

very few systems [4]. Like Tango, no in-session debugging tools are available in the EPPP.

PARSE

Unlike Proteus or Tango which uses a separate program to augment the compiled code, PARSE [5] 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. Performance debugging is available to analyze communication performance. However it is not user-friendly. The 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.

2.2.3 Functional Simulation System

In functional simulation systems, a parallel program is first translated into intermediate code of a virtual parallel machine (target machine). The set of intermediate code instructions is definable and can be different from the host's assembly language. Each instruction of the intermediate code is usually expressed as a host macro/procedure whose size depends on the complexity of the instruction and the desired level of simulation accuracy. At run time, the intermediate code instructions are interpreted at the functional level as if they were being executed on the target machine.

Functional simulation generally takes more simulation time than the direct execution approach. However, it is a very attractive technique for performance debugging due to its high accuracy and flexibility:

- In terms of accuracy, it is very accurate 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).

- In terms of flexibility, code interpretation permits the simulator to have complete control over program execution. This allows to establish the connection between user program code and the intermediate instructions. The user can thus set breakpoints, examine traced variables, or step through the program in a particular process. The 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 [1][10].

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 benefits 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 which may not be a natural structure to 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.

2.2.4 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, which 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 the 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 network simulator. It can simulate both packet switching and wormhole routing. CPSS can offer very accurate message routing and communication performance statistics for both routing models. Since wormhole routing technique is very popular on modern parallel computer systems, such as Intel Paragon [31], nCUBE6400 [32], and Ametek2010 [33], 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 Graphic User Interface

Graphic user interface has been designed for some of the parallel simulators to help the programmers to visualize and analyze program outputs and problematic performance statistics. METRICS [16] and Procsimity [17] are implemented using Tcl/Tk, which is an interpreted language. Applications witten in Tcl/Tk require an interpreter at run time and thus are slower. ParaGraph [18] is implemented using the X Window System. It is quite fast since it does not use any toolkit, but it does not take advantage of the latest developments in graphical user interface technology. U/IMX [19] is a toolkit for

developing graphic user interface based on X Window System. It has powerful capability to create GUI for existing application programs. However it is less flexible to integrate the graphic interface with a complex application program, such as CPSS. Java Abstract Window Toolkit [20] is another GUI development toolkit. Applications implemented in Java are portable across many platforms without modification. Again, Java is an interpreted language and thus applications written in Java are slow.

Our GUI for CPPE is developed using Motif toolkit [11] for the UNIX workstation platform and MS-MFC (microsoft foundation classes) [12] for the Windows PC platform. Motif is a set of guidelines that specifies how a user interface for graphical computers should look and feel. The term of Motif describes how an application appears on the screen (the look) and how the user interacts with it (the feel). Motif toolkit is developed by the Open Software Foundation (OSF) [34]. Compared with X window programming, Motif toolkit provides a higher level interface functionality and thus enables user to produce completely Motif-compliant applications in a relatively short amount of time. Compared with other X window toolkit, such as U/IMX, Motif toolkit provides the lower level programming functionality and thus provides the high flexibility to integrate application programs with the user interface. MS-MFC has become the most popular tools in writing Windows program. It provides a comprehensive set of classes for developing GUI objects and applications. Since it was written by the company that writes the Windows operating system, MFC is continuously updated to incorporate the latest changes to Windows itself.

Chapter 3

The CPSS System Architecture and Debugging Environment

In this chapter, we first describe our objectives for the design of CPPE performance debugger. Then we present an overview of the system architecture and high level design of CPSS, which provides the background information and creates the foundation for identifying the debugging environment and designing the performance debugger in CPSS. Then we present the high level design of the CPSS performance debugger.

3.1 Design Objectives

The final goal of the debugging system is to enable the user to interact with the underlying simulation system in order to better understand the behavior of parallel programs for correctness and performance tuning. To realize this goal, the most important objectives in the design of the debugging system are as follows.

- Correctness debugging: We should provide end-users with convenient debugging tools and useful information in order to debug and test applications, as in the case of sequential debugging environment. After all, this is the main advantage of using a simulator rather than a real parallel computer machine which provides a very limited set of debugging tools. Compared with conventional sequential debugging tools, the

challenge for designing the parallel debugging tools is that the tools should be able to trace individual parallel process during execution.

- Performance debugging: We should provide end-users with a set of performance monitoring tools which will provide computation and communication statistics of parallel application execution to facilitate the study of parallel architectures, network characteristics, parallel algorithms, program mapping and their effect on performance. Global statistics of an application (such as total execution time, speedup, total computation time and total communication time) allow the user to tune the applications to a desired performance. Run-time data or trace (such as process creation/termination information, message sending/receiving information) could be very useful in studying a particular aspect of the application, which can not be captured by global statistics.

- User friendliness: It should be easy to learn and use the simulator debugging tools. The functionality of each tool should be well defined and informative so that users can easily apply these tools in their programming exercises.

To meet the objectives of performance debugging, the following criteria should be followed in the designing of performance debugging tools:

- Informative: The information provided by the debugging tools should well address the functionality of corresponding performance affecting factors of parallel processing in the areas of both computing and network communication. From the performance statistics and run-time data, user should be able to well understand the

behavior of the parallel programs in terms of the relationship between performance and the changing of performance affecting factors.

- Accurate: The performance results obtained from the debugging tools should reflect accurately the behaviors of the parallel computer system. The results should be based on behaviors of realistic computation and communication models, yet making a good trade-off between simulation accuracy and simulation time.

- Repeatable: The results obtained from the debugging tools should be repeatable. Repeatability is essential to provide a stable debugging environment that is not available on a real parallel computer machine. Real parallel computers are nondeterministic in nature and, rarely provides any form of repeatability. Some bugs may not occur frequently enough for observation. Repeatability does not mean that the simulator can produce only one of the many possible executions of a nondeterministic application: the simulator should also be able to simulate multiple executions of an application when it is required to mimic the nondeterministic nature of real parallel computer systems and of parallel applications.

3.2 CPSS High-Level Design and Debugging Environment

This section gives an overview of CPSS high-level design, which provides the foundation for the design of debugging system in CPPE regarding the necessity and feasibility.

3.2.1 The General Structure of the CPSS

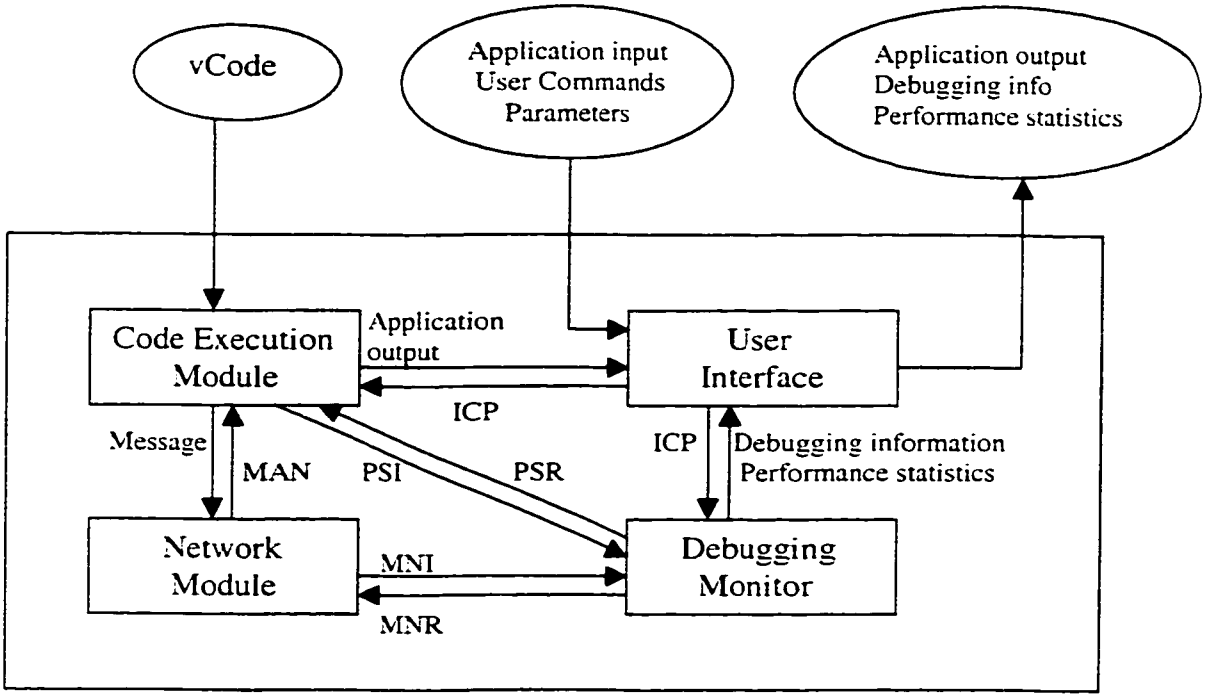
The CPSS (Concordia Parallel System Simulator) is an integrated part of the CPPE (Concordia Parallel Programming Environment). CPPE consists of two components: the CPCC and the CPSS.

The core of the CPCC (Concordia Parallel C Compiler) is a compiler. After reading a parallel program written in CPC (Concordia Parallel C) language, the CPCC builds a complete abstract syntax tree to perform syntax and semantics analysis, and produces object code for a generic virtual machine. Such object code is called *vCode* in CPPE. The vCode instruction set is defined based on an analysis of common operations of parallel computer systems. To produce vCode, the compilation process makes use of the virtual architecture and does not call for a physical architecture. The advantage of this design is that the CPC parallel program does not need to be re-compiled every time the underlying target architecture is changed.

The vCode produced by the CPCC will be input to the CPSS. Other inputs to the CPSS are parameters and commands from the user. For example, the user can specify the physical topology on which the program will run and the virtual-to-physical topology mapping. The CPSS then executes the vCode, using the parameters and commands entered by the user. The outputs from the CPSS are the application outputs, performance statistics, and debugging information.

The CPSS consists of two major components: the code execution module and the 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. The network module is to manage the inter-processor communication via message

passing. There are two other utility modules interacting with the code execution module and network module in CPSS: they are the debugging monitor and the user interface. The interactions between the components in CPSS are illustrated in Figure 3.



MAN: Message Arrival Notification
MNR: Message/Network Request
MNI: Message/Network Information
ICP: Input/Commands/Parameters
PSR: Process/Processor Status Request
PSI: Process/Processor Status Information

Figure 3 : CPSS structure and operations

The debugging monitor and interface not only closely interact with the two major components of CPSS during execution, their implementation is also dependent on the high-level design of these two components. In order to understand the design of the debugging monitor, we will first have an overview of the high-level design of code execution module and the network module.

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. There are three key issues that influent the design of the debugging monitor: simulation at the functional level, sequential execution model and the way timing system is implemented. CPSS uses the functional simulation technique which uses sequential execution model to emulate the parallel execution and interprets the parallel object code instructions at the functional level. This technique offers the most accurate results among the existing simulation techniques. In addition, this technique provides a good basis for performance debugging:

- Functional simulation: CEM interprets the intermediate parallel code at the functional level as if they were executed on the target machine. Each instruction of the target machine is usually expressed as a host macro or procedure whose size depends on the complexity of the instruction and the desired level of simulation accuracy. This technique permits the simulator to have complete control over program execution. It allows to establish the connection between user program statements and intermediate instructions. Thus the user can set breakpoints, examine trace variables, or step-through the program fragment of a particular process. Monitor code can be added to the simulating code without affecting the execution outcomes, because the CEM is able to distinguish between application code and monitor code and the execution time for monitoring code is not accumulated.

- By using functional simulation technique, we can parameterize system measurements (e.g. system clock cycle, execution time of object code instructions, network packet size, link buffer size, network delay, message and packet startup overheads). Performance statistics are based on these parameterized measurements.

- The sequential simulation is deterministic in nature. Therefore repetition of execution of a parallel program will always produce the same results and performance under the same system parameters. This provides a stable environment to study the performance of parallel programs at different levels of detail and from different perspectives.

- Timing system: CPSS does not use the machine clock for performance timing. There is a global clock for the simulated parallel computer system which is updated periodically by the CEM. Each process has a local clock that keeps track of the present time of this process. In the CPSS, parallelism is simulated by time slicing: each application process is given a quantum to run and processes are scheduled in a round-robin fashion. During each quantum, the job scheduler traverses the list of physical processors and schedules one process on a processor at a time for execution. The local clock of the scheduled process is updated after each instruction is executed. The user can define the cost to execute an instruction based on the complexity of the instruction. A process runs until its time quantum expires or it is put to sleep by some event. The job scheduler then schedules a process on the next processor for execution. When every processor has finished its quantum, the global clock is advanced to the

next quantum. By using this timing system, CPSS can provide accurate and repeatable performance statistics for performance debugging.

3.2.3 The Network Module

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

The following design issues of the network module is crucial for accurately simulating the communication behavior, yet providing feasibility for performance evaluation for parallel applications:

- By using the functional simulation technique and the same global clock mentioned in the CEM design, it can effectively simulate the network behavior and communication cost such as message startup overhead, routing overhead and congestion delay. New messages which are being initialized for routing are queued at a new message list. The waiting time at this list simulates message startup overheads. When the startup overhead time of a new message expires, the message will be removed from the list and appended to a list of active messages. In each quantum, all active packets that are not blocked are advanced by one link, and it simulates the movement of packets by advancing their ID numbers.

- Most of the network and communication parameters are well defined with appropriate data structures. User can configure most of the network parameters such as packet

size, flit size, routing scheme, link bandwidth, communication delay, network topologies and virtual-to-physical mapping without recompile of the simulator software and application programs, which provides a flexible performance debugging environment.

3.3 High-Level Design of Debugging Tools

Practical sources of performance degradation reviewed in Chapter 2 require the simulation system to provide the efficient and flexible debugging tools for parallel program development and performance fine-tune. The system architecture of CPSS provides a good debugging environment which is inherited from the functional simulation technique and the timing system employed in the design of the two major components (Code Execution Module and Network Module) in CPSS, as we reviewed in last section. We now identify the major debugging tools which will be implemented in CPSS and describe the high-level design of these debugging tools.

3.3.1 Correctness Debugging

Since 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. Unlike the case of direct simulation technique, the debugging code can be distinguished from the application code so that the amount of inserted debugging code does not affect the simulation results in terms of execution cost.

We identified and implemented the following functionality within CPSS for correctness debugging, with the design concepts borrowed from sequential programming environment:

- Set and clear instruction breakpoints: users can set breakpoints in the source program to automatically interrupt the program execution. Users can also clear breakpoints.

- Set and clear trace variables: users can set a trace flag on variables. Whenever the variable is referenced during execution, the program execution is suspended so that user can inspect the execution status. Users can also clear the trace flags.

- View the value of a variable: users can view the value of a variable in an active process when execution is suspended.

- Step through a process: users can suspend a program and then let the execution continue line by line or by a specified number of lines.

- Set a particular process to be the current process for debugging. The user may then use the above tools to debug the current process.

- View the program source code (written in the CPC language). Users can specify the range of the source code to be displayed.

- View the vCode corresponding to specified range of the source code.

- View the status of the processes. Information about each process includes
  - the processor on which this process is run

- the process status (e.g., ready, running, blocked, etc.)
- the stack of the process
- the function that is currently executed by this process
- the line in the source code that is currently executed by this process.

The main data structures related to correctness debugging are:

- Source code breakpoint table: CPSS maintains a global break table that stores all breakpoints set by user at run time. When executing in debugging mode, the code execution engine checks the line number of an instruction in the global break table before executing it.

- Trace variable table: CPSS maintains a global trace table that stores all traced variables set by user at run time. Since CEM references variables by their addresses in the memory pool, traced variables are stored in the trace table with their memory addresses. When a vCode references a variable, CEM will check whether the referenced variable is in the trace table.

- Source code and vCode table: Source code table stores the application source code, and vCode table stores the compiled virtual machine code. CEM executes the vCode in the vCode table, with the index of the vCode table servers as the program count (PC).

- Source-to-vCode table: each source code is usually compiled into several vCode instructions for execution. The source-to-vCode table src2codTable will associate the source line number with the vCode line number so that we can trace the program

execution. The first vCode line number of each corresponding source code line will be stored in this table.

- Memory pool: Memory blocks will be allocated from the memory pool and distributed to running processes for program execution. Memory pools is implemented as a fixed-size array, with the index of the array serves as the memory address. Variables are accessed by their addresses in the memory pool.

3.3.2 Performance Debugging

Based on the review on the sources of performance degradation in Chapter 2, we not only need 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. To study the communication overhead of parallel computing, we need to simulate a variety of network topologies and be able to easily configure the communication parameters. To study the optimal program mapping from virtual architecture to physical architecture, we need to provide the user with the ability to specify certain patterns of virtual-to-physical-architecture mapping.

We identified the following performance statistics for performance debugging and implemented corresponding functionality within CPSS to provide these information:

- Set and clear time breakpoints: user can set an alarm to automatically suspend the program execution when a certain time is reached. When program execution is suspended, user can query various performance data and statistics. User can also clear the alarm.

- Parallel execution time: the estimated execution time of the program run on an actual target multicomputer or multiprocessor.

- Sequential execution time: the estimated execution time of the program run on a uniprocessor computer.

- Execution time of any portion of the program, either sequential or in parallel

- Computation time of the program: time the program spent on computation task

- Communication time of the program: overhead involved in inter-processor communication such as message sending/receiving, congestion delay

- Processor utilization

- Profile of processor utilization as a function of time

- Process creation/termination information, such as time, processor number, parent process ID

- Message send/receive information, such as time, source node, destination node, message length

- Message routing information, such as path, time traces

User can reconfigure the network by specifying different topology, virtual-to-physical mapping and redefine most of the communication parameters in order to study the performance at different levels from different angles. We also provide the utility to

log these performance statistics data into files that can be retrieved later for further analysis as the user wishes.

The main data structures related to performance debugging are:

- The global clock: CPSS uses relative timing with a user defined clock cycle. The global clock is advanced to next quantum only when every application process has finished its quantum. Global clock simulates the actual elapsed execution time on a real parallel machine. Both parallel and sequential execution time are accumulated based on this global clock.

- Process local clock: each parallel process maintains a local clock using the same clock cycle as the global clock. During parallel execution, each parallel process is given a quantum to run until its quantum expires. The process local clock is used to keep track of the time spent during the given quantum.

- Network and communication parameters: network and communication parameters are all configurable variables defined in CPSS. The network architecture is defined by network type, network dimension and size of each dimension. Communication parameters are defined in a parameter structure. Different network architectures and communication patterns can be simulated by simply redefining corresponding variables or parameters.

- Virtual-to-physical mapping table: this table is used to store the data that reflect a specific pattern of virtual-to-physical-architecture mapping. Using this table at run-

time, a virtual processor will be mapped to a physical processor on which a parallel process will be running.

3.3.3 Graphical User Interface

Due to the complexity and multiple dimensionality of parallel performance data, we provide a graphical user interface (GUI) to help programmers in their development process. The user interface enables the user to interactively communicate with the simulator. The user interface receives parameters and commands from the user, validates the received information, and passes valid parameters and commands to the appropriate module (the CEM, the network module or the debugging monitor). During execution of a parallel program, the user interface is used to interact with debugging monitor and display performance statistics and debugging information. Program outputs are also transferred from the CEM to the user interface for displaying.

Currently, CPPE runs on UNIX workstations and MS-Windows PCs. The GUI for UNIX workstations is developed with MOTIF toolkit [11], and the GUI for MS-Windows PCs is developed with MS-MFC (Microsoft Foundation Classes) [12].

To modularize the development, the interface code is isolated from the other components in CPPE as much as possible. Since CPPE is continuously 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.

Chapter 4

The Design of the Debugging Monitor

This chapter presents the design and implementation of the debugging monitor. The debugging monitor is composed of a set of debugging tools. We divide the debugging tools into two major categories: correctness debugging tools and performance debugging tools. In first section we present the design of correctness debugging tools. In the second section we present the design of performance debugging tools. In the second section. we also describe the design of performance statistics as part of the program output.

4.1 Correctness Debugging Tools

In chapter 3. we have identified a set of useful tools for correctness debugging. Basically, these tools are used for two types of purpose:

- Tools for execution control using which the user can step through the program execution. These include tools for setting and clearing source code breakpoints, and tools for stepping through program execution. In order to design and implement the tools for execution control, we need to understand how a vCode program is executed and how execution proceeds in CPSS in order to set breakpoints during execution. This concerns the mechanism of the execution function of the code execution module

(CEM). We will review the execution mechanism of CEM in the design of relevant tools that follows.

- Tools for execution examination using which the user can trace the execution status. These include tools for viewing source code and vCode, tools for tracing and viewing variables, and tools for examining the status of process and process stack. To design and implement the tools for examination purpose, we need to understand how data and variables are stored in the execution environment in order to retrieve them at run time. This concerns the memory management of CEM. We will review the memory management of CEM in the design of relevant tools that follows.

4.1.1 Setting and Clearing Source Breakpoints

Functionality

We implemented two tools in this category: Break and Clear Break. Break function is used to set breakpoints to interrupt execution so that the user can step through a particular parallel process and trace variables in that process. Before the execution of a vCode file or after each breakpoint suspension, the user can set breakpoints by referring to the line numbers in the source program. At the same time, user can also clear previously set breakpoints by referring to the line numbers in the source program. A breakpoint is assigned to a specific line of source code in the program. When any running process tries to execute a line in the program with a breakpoint, the whole program execution will be suspended, including the execution of all the processes.

As we will discuss in the following section, the implementation of Breakpoints is based on the vCode of a compiled source program. Since the vCode is produced only for the executable source code in the compile process, therefore only executable source lines can be selected as breakpoints.

System Status

In order to control the execution of the application program, CPSS defines a set of system states and the behavior of CPSS in each system state. It also defines the rule of transformation between different system states. The major system states include Run, Break, Halt, Dead, and various kinds of error states.

When the system state is set to Run, the Code Execution Module (CEM) in CPSS will execute the program until the system state becomes a value other than Run. The system state is set to Run when one of the following conditions is true:

- when CPSS execution is first started
- when execution is resumed from a breakpoint by Continue function in the debugging tools
- when execution is resumed from a breakpoint by Step function in the debugging tools

When the system state is set to Break, the CEM will suspend the program execution. A suspended execution can be resumed and continue execution from the last suspension point. The system state is set to Break when one of the following conditions is true:

- when a breakpoint set by a user is encountered during execution
- after the execution of one step of source code when using Step function
- a traced variable is referenced
- the alarm time set by the Alarm function in the debugging tools is reached

The system state is set to Halt when a user program executes the vCode HALT when it finishes execution successfully so that the execution session can terminate normally. When the main process in a user program finishes execution, it executes the vCode HALT to signify the end of execution. The main process then waits for all its forked child processes to terminate. After all forked child processes terminate, the main process terminates itself and sets the system state to Halt.

The system state is set to Dead when fatal problems happen during program execution. As an example, all processes are deadlocked so that no process can be scheduled to run. A Dead execution session is unrecoverable and users must restart the execution.

Data Structure of Breakpoint Table

CPSS maintains a global breakpoint table that is used to store all breakpoints set by the user at run time. The breakpoint table is implemented as an array **breakTable**, and each breakpoint is implemented as a structure of **breakEntry**, defined as in Figure 4:

```
struct    BreakEntry
{
     int  line;          // source code line number
     int  memLoc;        // vCode line number
};

BreakEntry     breakTable[MaxNbrBreaks];
```

Figure 4 : Data structure of Breakpoint table in C

In the structure **BreakEntry**, the field **line** is the line number in the application source code, and the field **memLoc** is the line number of the vCode. The line number of the vCode is used by CEM as the process count (PC). Since CEM executes a program based on the vCode rather than source code, we should associate each breakpoint in the source code with the corresponding vCode line number.

Implementation

When a breakpoint is set at a specified source line, this source line will be insert into the global breakpoint table with its source line number and its first corresponding vCode line number, the **memLoc**, in the structure of **BreakEntry**. The first vCode line number of a source code can be found from another global table maintained by CPSS: **src2codTable**. Every time the CEM increments a current process's PC, it checks the break table before it executes the code. If the value of the PC is found in the breakpoint table by satisfying the condition:

```
currentProc->PC == breakTable[i].memLoc
```

the system state will be set to Break and the execution is suspended at this point. A flag is used to set the debugging mode On or Off during execution. CEM checks the breakpoint table only when the debugging flag is set to On while ignoring the breakpoints when the debugging flag is set to Off. During program development, user can set the debugging flag On in order to use the debugging functions. In real execution of the program, user can set the debugging flag Off to make the execution fast.

To clear a breakpoint, simply delete the breakpoint entry from the breakpoint table.

4.1.2 Stepping Through a Process

Functionality

When any parallel process tries to execute a line in the program with a breakpoint, the whole program execution will be suspended. From that point, the execution of the program may be continued in two manners: either continuing the execution until the next breakpoint is encountered by any process, or user can follow the execution line by line. We implemented two tools for this purpose: Continue and Step. By using Continue, the program will resume execution from the breakpoint and continues until the next breakpoint is encountered or the execution finishes. By using Step, the default action is that the program will continue the execution from the breakpoint line by line upon each Step call. The process where the suspension occurred becomes the current "Step-Process". User can also override the default action by specifying a different *Running* process as current "Step-Process" or specifying a different number of lines in each step.

(To get a list of *Running* process, we need another tool called Status which displays the status of all active processes. The design of the tool Status will be introduced later.)

Data Structure

CPSS has two global variables that are used by CEM to control the execution starting and ending point when stepping through a process: startCodLineNbr and endCodLineNbr. During the execution of a source line, startCodLineNbr stands for the starting vCode line number for the source line, and endCodLineNbr stands for the ending vCode line number for the source line. CEM uses these two variables to determine the execution starting and ending points when CEM is invoked to execute one source line. CEM has a local variable that keeps track of the current execution point: curCodLineNbr which stands for the currently executed vCode line number. When CEM steps through a process, curCodLineNbr should be always between startCodLineNbr and endCodLineNbr inclusive.

CPSS has another global variable curProcess, which stands for the currently scheduled process. It is defined as a pointer to structure of the process, which contains all run-time information of a parallel process during execution. One of the information is the process count (curProcess->PC), which keeps track of the current execution point in the vCode table.

Implementation

By default, we set the currently scheduled parallel process when program is suspended to be the "Step-Process", which is pointed to by a global variable steppingProc in CPSS.

During stepping, curCodLineNbr increments only when the stepping process is rescheduled to run. All other parallel processes are scheduled and run in the same way as normal execution. So essentially we are stepping through a particular parallel process. Both Step and Continue function will resume the execution of CEM from the last breakpoint, which is the process count of the currently scheduled process (curProcess->PC). For the Continue function, CEM restarts its execution on the vCode from the resuming point and continue until it encounter the next breakpoint if any or the end of the program. This can be implemented by simply setting the system state to Run and recalling the CEM function execute().

To implement the Step function, we need to tell the CEM the starting and ending point in the vCode corresponding to one source line. The starting point is the first vCode corresponding to the source line which is to be stepped, and the ending point is the last vCode corresponding to the same source line. This is implemented by using the global variables startCodLineNbr and endCodLineNbr. Every time Step function is called, the system state is set to Run, the startCodLineNbr is set to the first vCode of the source line which is steppingProc->PC, and the endCodLineNbr is set to the last vCode of the same source line, which can be calculated based on the src2codTable:

```
startCodLineNbr = steppingProc->PC;
endCodLineNbr = src2codTable[breakLine + 1] - 1;
```

where breakLine is the source line number of the breakpoint. CEM function execute() is then called to execute the vCode between the startCodLineNbr and endCodLineNbr inclusive. execute() function maintains a local variable curCodLineNbr, which increments at the same pace as the process count (steppingProc->PC). When

curCodLineNbr becomes greater than endCodLineNbr, the system state is set to Break, and the program is suspended again, which concludes the execution of one line of the source code.

We can easily extend the Step function so that it can step through any running process when program is suspended. Simply define any running process to be the "Step-Process" by assigning it to be steppingProc, recalculating the values for startCodLineNbr, endCodLineNbr and curCodLineNbr, and curCodLineNbr increments only when the newly assigned stepping process is rescheduled to run.

By default, the number of lines in one step is one. We can also extend the Step function to be able to step through any number of lines in one step. Another global variable stepNbrSrcLine, which stands for the number of source lines in one step, is needed for this purpose. CEM executes the first source line in the step as we described before, however, instead of setting the system state to Break after finishing the first source line in the step, it recalculates the startCodLineNbr and endCodLineNbr for the second source line in the step, and decrements the stepNbrSrcLine, and executes the second source line in the step. The system state will be set to Break only when stepNbrSrcLine decrements to 0, and at this point the program will be suspended, which concludes the execution of one step of a specified number of source lines.

4.1.3 Viewing Source Code and vCode

Functionality

There are two tools in this category: List Source and List vCode. List Source is used to list a fragment of the application source file. List vCode is used to list a fragment of

vCode of an application. Before the initial execution or after each execution suspension, the user can specify a range in the application source file, the List Source function will display the source code within that range, and the List vCode function will display the vCode instructions corresponding to the specified range of source code. If no line numbers are specified, the whole source file or vCode file will be displayed.

Data Structure

Three tables are involved to implement this functionality: srcTable, codeTable and src2codTable. srcTable is used to store source code, codeTable is used to store execution code (vCode), src2codTable is used to associate the source code and the vCode. Before a program is executed, there is a loading process during which a set of global tables are loaded from the compiled vCode file. srcTable, codeTable and src2codTable are three of these global tables. srcTable is loaded with source code, with the index of the table stands for source line number and entries of the table are the actural source code. codeTable is loaded with vCode. The index of codeTable stands for the vCode line number which serves as the process count (PC) during execution, and the entry of the codeTable is the vCode that is defined as a structure of virtual machine code and its operands. One source code may involve several successive vCode instructions, so a src2codTable is used to associate the source code and vCode. The index of the src2codTable stands for the source code line number, the entry of the table is the line number of the first vCode instruction generated from that source code line.

Implementation

To list the source code, simply display the contents of the srcTable from the starting point to ending point specified by the user.

To display vCode, we think the following information will be helpful for the user, and these information can be retrieved from the codeTable and src2codTable: the source line number, the corresponding vCode line number, the vCode name, operands of the vCode instruction. All these information will be displayed in one line for each vCode instruction, and the range of vCode instructions to be displayed is determined by the user specified range of source code and the src2codTable, which provides the first vCode instruction line number for each source code line number.

4.1.4 Tracing Variables

Functionality

Whenever the execution of the parallel program is suspended, the user may want to examine the current value of variables in the current execution environment of each process. We implemented two functions for this purpose: Show and Trace. Function Show is used to display the value of a variable when program execution is in suspension state. Users can use this function to examine variables in the current environment of each active process. Active processes are those that may be in the state of Ready, Running, Blocked, Delayed or Spinning. We developed another tool called Status which displays the status of all active processes.

Function Trace is used to trace a particular variable during the execution process. The user essentially sets a flag on the traced variable. Whenever that variable is referenced during subsequent program execution, the program will be suspended as it is

for breakpoints. In addition, the program will display the value of the traced variables so that the user can examine the status of the program execution. Since variables may be referenced by many different processes in a variety of locations, the trace function provides a useful tool for the user to focus on some important program variables.

In order to implement the Show and Trace function, we first need to understand how CPSS manages local memory to simulate a parallel execution and how variables are stored in such simulation environment. Then we can easily trace and retrieve values of variables during program execution.

Memory Management of CPSS and Data Structures

In the CPSS model, parallel processes time-share a physical processor, thus the corresponding local memory is shared by many processes. When a new process is created on a processor, a memory block is allocated from the local memory for that process (Figure 5a), which is used for the following types of data:

- working stack: the stack is needed for expression evaluations and for temporary run-time data.

- activation records: each record contains function parameters, local variables inside the called function, and other control information for the function call/return. An activation record is allocated on every function call, and de-allocated on the function return.

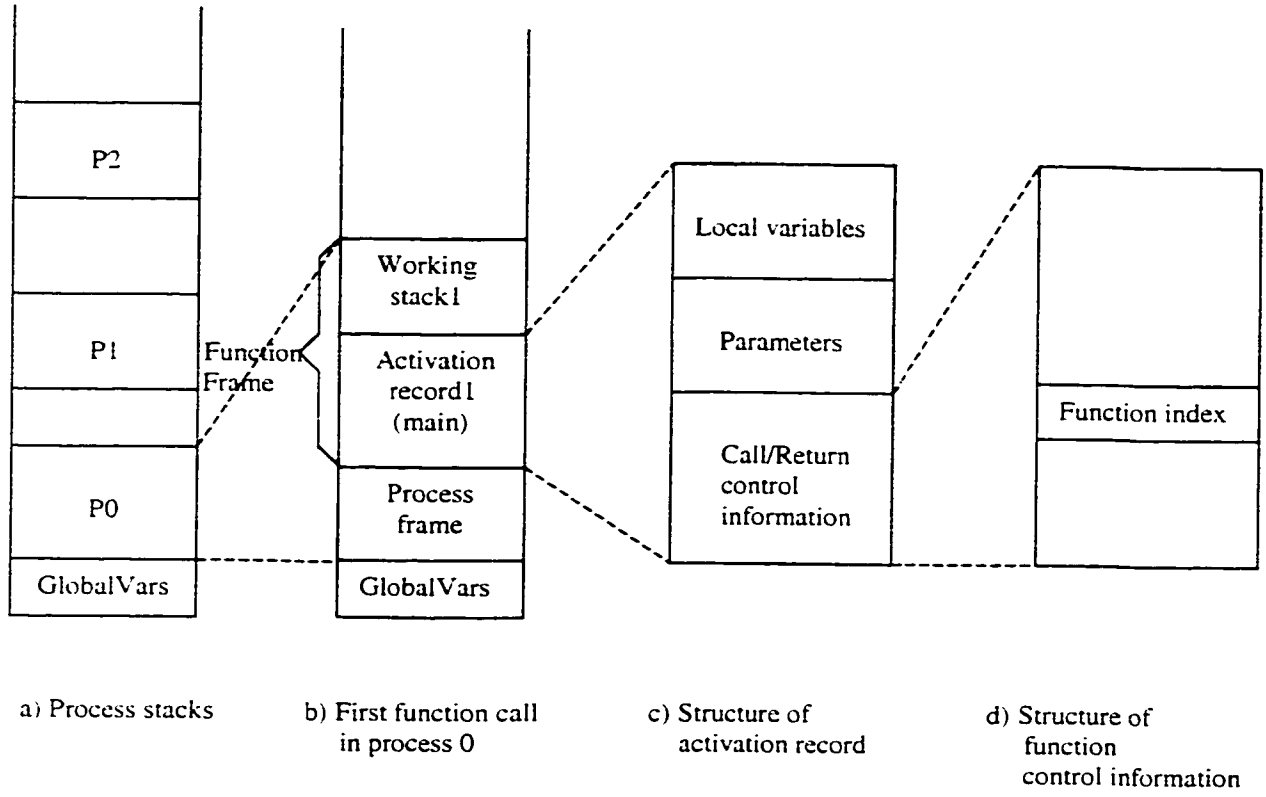


Figure 5 : Memory management of process stacks

When a process is created, the very first working stack granted to it is called a process frame. Every time a function is called from a process, a function frame consisting of an activation record and a working stack is allocated to the calling process (Figure 5b). If a second function is called from inside the first function, a second function frame will be allocated from the local memory and linked to the first function frame. When a function returns, the corresponding function frame will be de-allocated.

The process created when program execution starts is the process 0. Global variables are stored starting at address just below the first process frame for process 0 (Figure 5b). Local variables inside a function are stored in the activation record of the

corresponding function frame (Figure 5c shows the activation record for main()). The name of the function is indexed in a global table of identifiers, with the index stored in the function control information sector of the activation record (Figure 5d).

Several major data structures are used to implement the above memory management in CPSS, which are also crucial in the implementation of our debugging tools:

- Memory pool: Memory blocks are taken from a common *memory pool* and distributed to requesting processes. CPSS implements the memory pool as a fixed-size array, the **storageValue[]**. The entry of the array is considered to be a memory word and is implemented as a C Structure type indicating the data type and the data value (Figure 6). An array parallel to array **storageValue[]** is used to store locations (physical processor IDs) of a memory word, the **storageOwner[]**, so that CPSS can know to which physical processor each entry in the memory array belongs.

- Process Control Block (PCB): every application process is associated with a process control block (PCB) that stores various information needed for its execution. When a new process is created, a PCB is allocated and appended to a global list of processes which is not implemented in the memory pool. This is a singly-linked list managed by three pointers: **actProcHead** pointing to the first entry of the list, **actProcTail** pointing to the last entry of the list and the **curProc** pointing to the PCB of the process currently running. As a process is running, its memory address in the common memory pool is maintained and handled by four pointers which are stored in PCB, they are **base**, which points to the starting address of the process stack

allocated when the process is created; **B**, which points to the starting address of the current function frame; **T**, which points to the stack top of the current working stack; and **stackTopLim**, which points to the upper bound of the current working stack.

```
typedef struct {
     char type;
     union {
          int intValue;
          float floatValue;
     } val
} basicValue, *basicValuePtr;

basicValue storageValue[STORAGE_SIZE];

int storageOwner[STORAGE_SIZE];
```

Figure 6 : Data structures of memory pool in C

- Global tables: CPSS uses several global tables outside the memory pool to store different run-time information. Among them, the important tables that are related to the debugging tools are **identTable**, which stores the identifiers such as function names and variables; **blockTable**, which has one entry for each compound statement (function body is also a compound statement); **arrayTable**, which stores information of arrays; and **floatConsTable**, which stores the constants of float type.

Implementation of Show Function

Based on the memory management and data structures introduced above, we now introduce how to retrieve a variable value in a specific application process. This is the

basis for implementing the Show debugging tool. In CPSS, different types of variables have different ways of storage, therefore the ways to retrieve their values are also different. The following sections introduce the Show function for variables of major data types supported in CPC.

1. Show Function for data type of integer, float, char

Figure 7 shows a typical process to get the value of a local variable of type integer, char or float inside a currently called function in a specific application process.

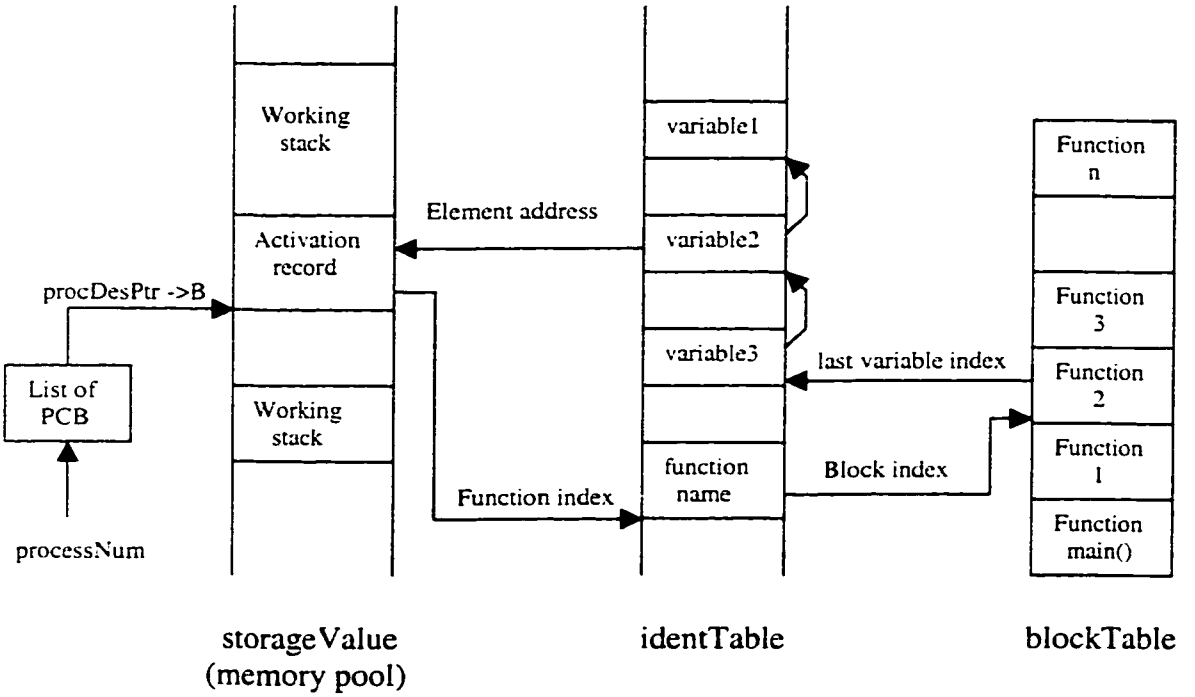


Figure 7 : The process of retrieving the value of a variable

The prerequisites for the process in Figure 7 are:

- The variable is of type integer, char or float.

- It is a local variable of a function that is currently called by an application process.

- The function is not the program's main() function. (Process for variables in main() function will be discussed later.)

- The application process must be an active process defined as before.

- The input of the value retrieving process are the process number and the variable name.

The value retrieving process can be depicted in the following steps:

- Get the process description pointer (procDesPtr): according to the process number specified by the user, we first get the pointer to the process description block for that specific process by searching the global PCB list.

- Get the function index (funcIndex): as described before, each currently called function is allocated an activation record and a working stack, and one of the field in the activation record is the function index which is a pointer to an entry in the identTable corresponding to this function identifier. The PCB has a field called B which is the pointer to the starting address of the current function frame, therefore the function index can be calculated by the formula

```
funcIndex = ProcDesPtr->B +
     offset of function index in activation record
```

- Get the block index for the function: the function identifier has an entry in the global identifier table **identTable** indexed by the **funcIndex**. One of the information of the identifier entry is a reference (**identTable[funcIndex].ref**) to an entry in another global table called **blockTable**. **blockTable** is used to store information about compound statements including function bodies, with one entry for each compound statement.

- Get last variable index of the function: local variables in a function are maintained in the identTable using a linked list in reverse order, with the index of the last variable stored in the function entry in the **blockTable**. The index of the last variable (**blockTable[idenTable[funcIndex].ref].lastIdent**) is the index of the variable in the identTable.

- Get the offset address of the specified variable in a function frame: every local variable of a function has an entry in the **identTable** and they are related by a linked list. Starting from the last variable, we can search for the specified variable in the variable linked list and get its address (**identTable[n].address**) for a particular index value **n**. The address **identTable[n].address** is the offset of a variable in a function frame where the value of that variable is stored. Since CPC supports nested function definition, users can specify a variable at different levels of function scope. Therefore we start the search from the highest scope level (the most nested function scope) i to the lowest scope level 0 (for global variables) until the user specified variable is found, and the scope level **scopeLev** will be used to calculate the base address for that function frame.

- Get the address of the specified variable in the memory pool (**elemAdd**): the actual address of a variable in memory pool (**elemAdd**) is the sum of its offset address in the function frame (**identTable[n].address** in last step) and the base address (**B**) of the function frame in the memory pool. The process description block has a data field display[] that stores the base addresses of each function frame regarding that particular process. For each integer $i \geq 0$, display[i] stores the base address of the function frame where the function is declared at scope level i. Having known the scope level (**scopeLev** from last step) of the function that contains the user specified variable, we can get the base address of the function frame, **procDesPtr->display[scopeLev]**. Therefore **elemAdd** can be calculated by the formula

```
elemAdd = identTable[n].address +
      procDescPtr->display[scopeLev]
```

- Get the value of the variable: during function execution, the value of a variable is kept in the activation record of that function frame and updated with the proceeding of the program execution. Therefore, we can display the value of a variable by referring to the corresponding storage in the memory pool as

```
storageValue[elemAdd].val.intValue
                  // for integer or char variable
      or
storageValue[elemAdd].val.floatValue
                  // for float variable
```

For a global variable, the value retrieving process defined above will be simplified, as global information is always maintained in process 0, and the entry in blockTable for the imaginary block containing all the global variables is always the first one (blockTable[0]). Global variables are reversely linked together in the same way as a function's local variables, with the index of the last variable defined in blockTable[0].lastIdent. In the same way as described above, we can get the address of the specified global variable in the memory pool and display the value during execution.

2. Show Function for Data Type of Structure

The value retrieving process for a Structure variable can be depicted in the following steps:

- First, we use the same approach as we described before for data type of integer, float and char to get the entry for the structure variable in the identTable (identTable[n]) (Figure 7). Each entry for a variable in the identTable has a field to indicate the data type of this variable as well as its compound level and offset of the variable in a function frame. If the variable is of structure type, the offset address in the entry structure (identTable[n].address) is actually the offset address of the first field in the structure variable, and another field in the entry structure called ref will provide the index (identTable[n].ref) for the first field of this structure variable in the identTable (field 1 in the identTable, Figure 8).

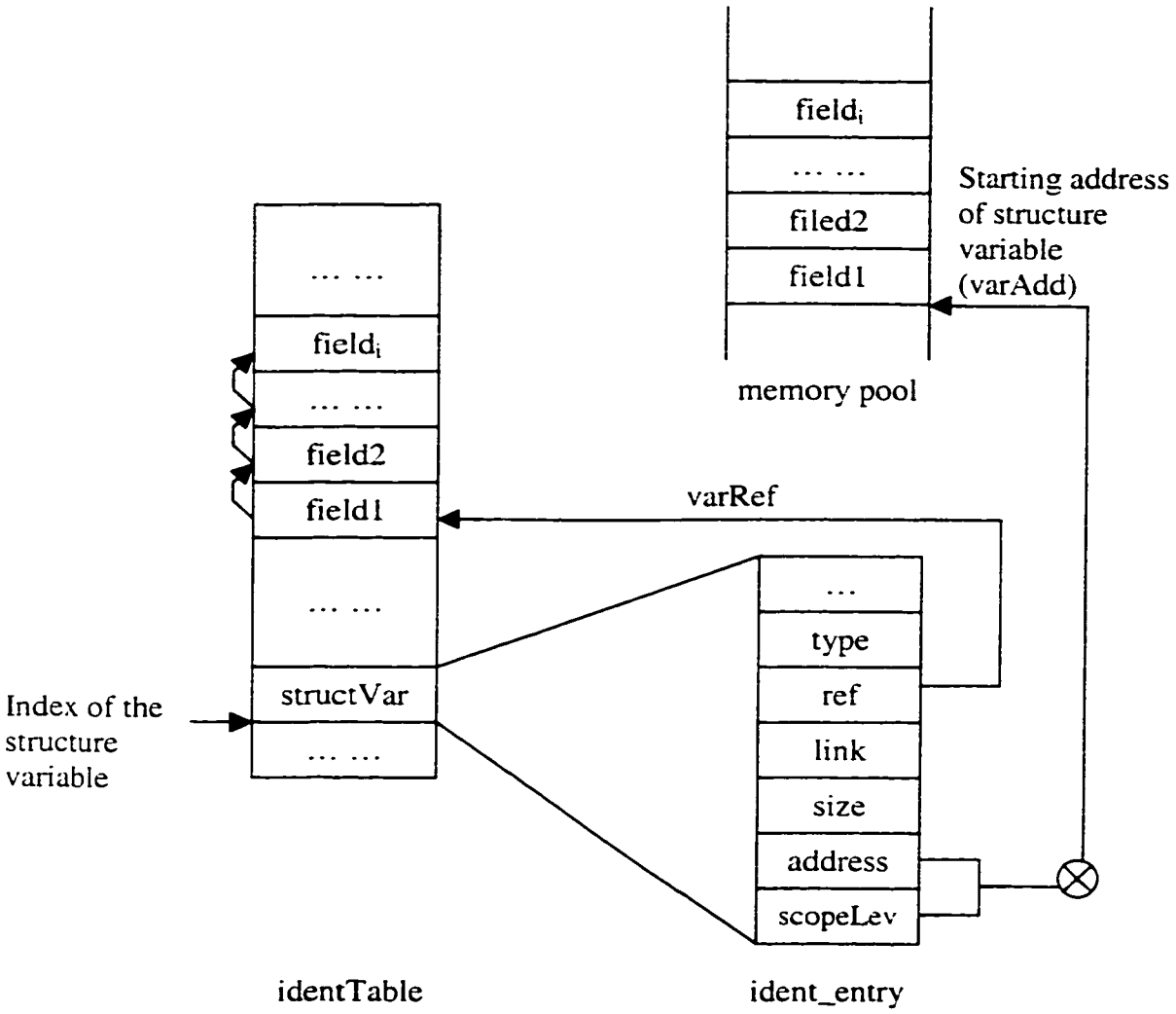


Figure 8 : The process of retrieving the value of a structure variable

- Data fields in a structure variable could be of any data types. Each field has a ident_entry structure in the identTable. The size field of the ident_entry structure (identTable[m].size) records the corresponding size of the field data type, and the

link field of the ident_entry structure (identTable[m].link) gives the index of the next field of the structure variable in the identTable.

- We need to get the address of each field in the structure variable in the memory pool in order to retrieve the value of each field. To get the address of the first field in the structure variable in the memory pool ($elemAdd_1$), we can use the offset address of the structure variable in the function frame (identTable[n].address) and the base address of the function frame (procDescPtr->display[scopeLev], as we described before for data type of integer, float or char:

$$elemAdd_1 = identTable[n].address + procDescPtr->display[scopeLev]$$

- Since all the fields in a structure variable are contiguously stored in the memory pool, the address of the second field will be address of the first field moving forward by the size of the first field. In general, the address of the field (i+1) ($elemAdd_{i+1}$) can be calculated by the following formula:

$$elemAdd_{i+1} = elemAdd_i + identTable[i].size$$

where identTable[i] is the entry for the field *i* in the structure variable in the identTable (Figure 8).

- If all the fields in the structure variable are of type integer, float or char, we can create a ShowElement() function, based on what we described before for the data type of

integer, float or char, and go through all the fields in the structure variable to display their values.

- If a field in the structure variable is a compound data structure, a recursive call to the function ShowElement() will be required to process all the fields in that structure field. By using recursive function call, we can generalize the Show function for any data type of the fields in the structure.

3. Show Function for Data Type of array of integer, float, char

To implement Show function for data type of array, we need to use another data structure called arrayTable. Each array used in the application program has an entry in this table. The entry structure of this table stores information about an individual array such as index range (low, high), element type (type) and element size (size).

The value retrieving process for an array can be depicted in the following steps:

- First, we use the same approach as we described before for data type of integer, float and char to get the entry for the array variable in the identTable (identTable[n]) (Figure 7). The entry structure in the identTable has a field to indicate the data type of this variable. If the variable is of type array, the offset address in the entry structure (identTable[n].address) is actually the offset address of the first element in the array variable, and another field in the entry structure called ref will provide the index (identTable[n].ref) for this array in the arrayTable (Figure 9).

- From the arrayTable, we then get the array index range (arrayTable[m].low, arrayTable[m].high), element size (arrayTable[m].size) and element type (arrayTable[m].type).

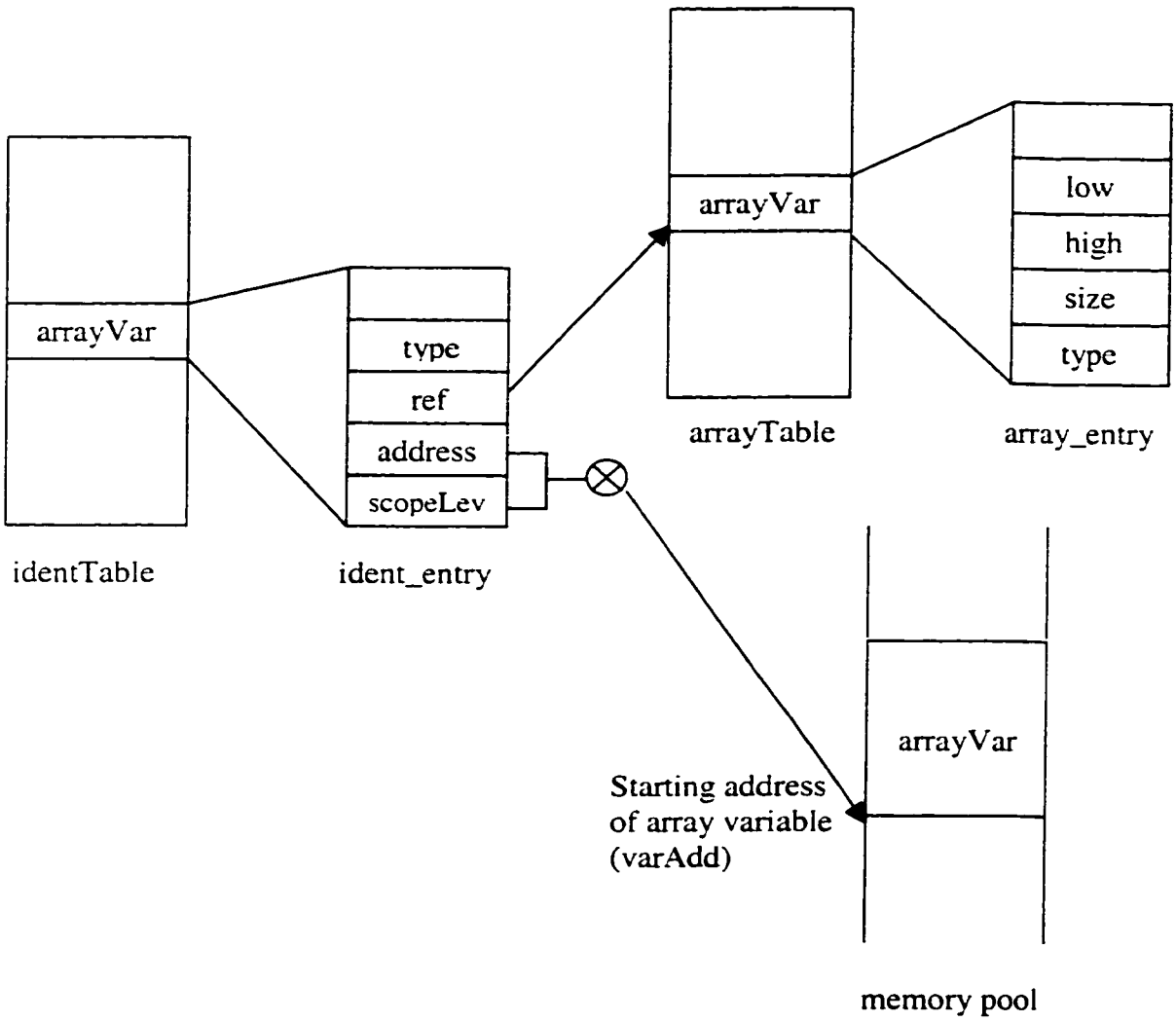


Figure 9 : The process of retrieving the value of an array variable

- We need to get the address of each element in the array variable in the memory pool in order to retrieve the value of each element. To get the address of the first element in the array variable in the memory pool ($elemAdd_1$), we can use the offset address of the array variable in the function frame (identTable[n].address) and the base address of the function frame (procDescPtr->display[scopeLev], as we described before for data type of integer, float or char:

$$elemAdd_1 = identTable[n].address + procDescPtr->display[scopeLev]$$

- Since all the elements in an array variable are contiguously stored in the memory pool, the address of the second element will be address of the first element moving forward by the size of the array element. In general, the address of the element (i+1) ($elemAdd_{i+1}$) can be calculated by the following formula:

$$elemAdd_{i+1} = elemAdd_i + arrayTable[m].size$$

where arrayTable[m] is the entry for the array variable in arrayTable (Figure 9).

- If all the elements in the array variable are of type integer, float or char, we can create a ShowElement() function, based on what we described before for the data type of integer, float or char, and go through all the elements in the array variable to display their values.

- If an element in the array variable is an array, then we are dealing with a multi-dimensional array. In that case, a recursive call to the function ShowElement() will

be required to process all the elements in that secondary dimensional array element . By using recursive function call, we can generalize the Show function for variables of multi-dimensional array.

4. Show Function for Channel Variables

Channel variables are specific for CPC programs and write/read operations on channel variables abstract message send/receive in the real parallel computer. A channel can be considered as an infinite buffer owned by some process p. where other processes can deposit messages of the same type as the channel for p to read. From the perspective of debugging, channel variables can be treated as other application variables in the sense that user can inspect the contents of the channel buffer in order to trace the message flowing during parallel execution.

Control information of channel is stored in a structure called *channel descriptor.* When a channel is opened (i.e. accessed for the first time), a channel descriptor is allocated for this channel which contains the data fields shown in Figure 10.

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 arriving time, and maintained in a list for reading by the channel owner. This list will be referred to as the *list of channel values.* The field **head** in the **Channel** structure points to the first element of the list. The Show function for a channel variable is meant to display the *list of channel values* maintained in a specified channel variable at a program suspension point.

Channel descriptors are taken from an array of channel descriptors, array **chann[]** (Figure 10), when a channel variable is first time accessed. As other variables, a channel variable also has an entry *e* in the memory pool (array **storageValue[]** in Figure 6). The index of this entry in the menory pool is called the *address* of the channel variable. Unlike a normal variable such as an integer or a float, a channel variable does not have a specific value. It represents a list of values (messages). Before the channel is opened, entry *e* was initialized to 0, meaning that the channel is currently inactive. When the channel is accessed for the first time (either for read or write), a channel descriptor is allocated to the channel. The index of the channel descriptor in the array of channel descriptors **chann[]** is called the **channel ID**, which will be stored in entry *e* of the memory pool.

```
typedef struct
{
     int head;
     // pointer to the head of the list of channel values
     int dataCount;
     // number of elements in list of channel values
     ProcessNodePtr waitProcQueue;
     // pointer to blocked reader's PCB
     int chanElemSize;   // message size
} Channel

Channel chann[MAX_NUM_CHANNELS+1];
                          // array of channel descriptors
                          // entry chann[0] is unused
```

Figure 10 : Data structure of Channel variable in C

The value retrieving process for a channel variable can be depicted in the following steps (Figure 11):

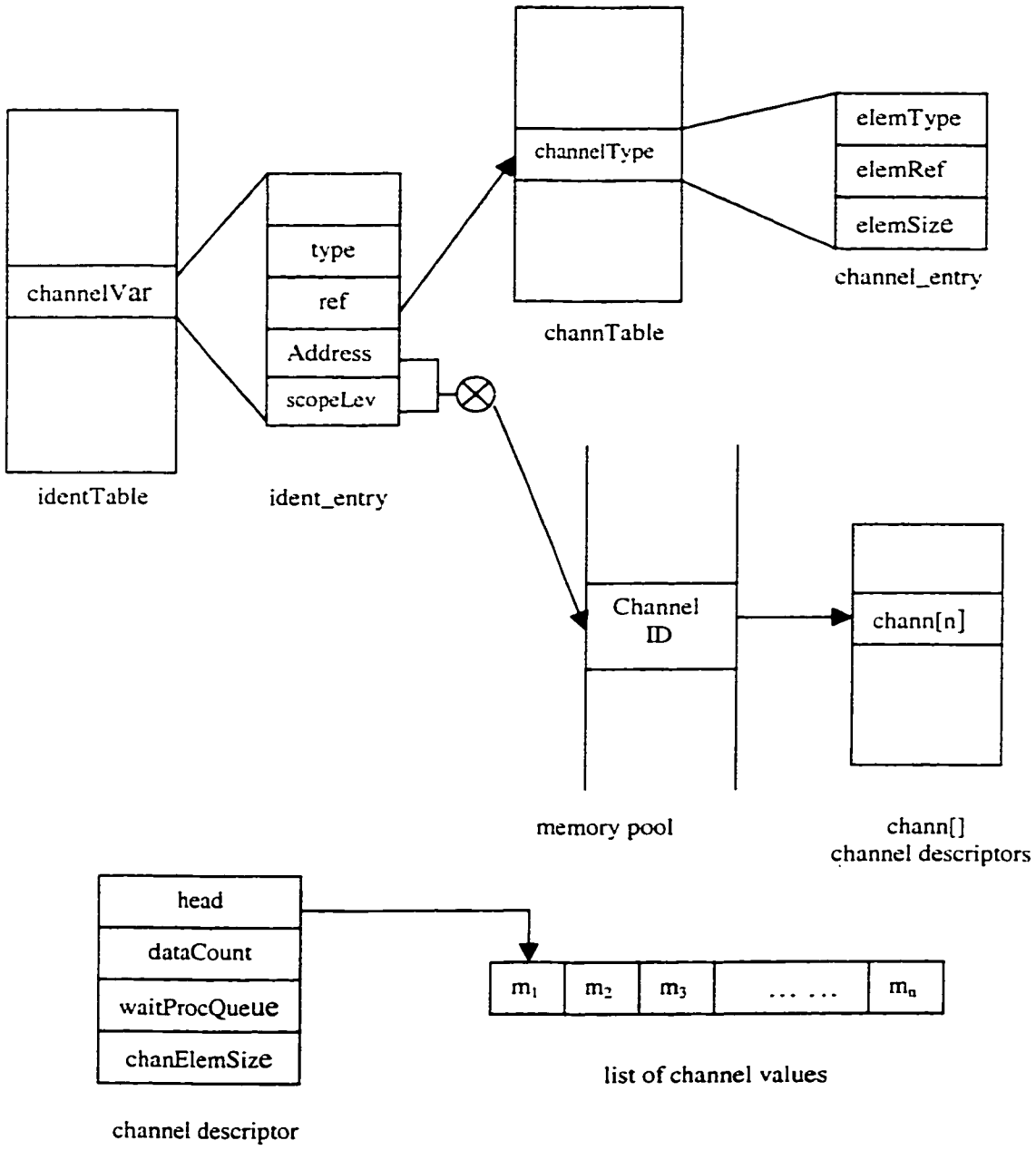


Figure 11 : Retrieve messages from a Channel variable

- First, we use the same approach as we described before for data type of integer, float or char to get the channel variable in the identTable (Figure 7). For a channel variable, the ref field in the ident_entry structure provides the index (identTable[n].ref) for this channel variable in a global channel table channTable. channTable stores the information about each channel type such as the message type and message size for this channel.

- From the channel address (identTable[n].address) and the base address of the function frame (procDescPtr->display[scopeLev]. we can access the entry of the channel variable in the memory pool and obtain the channel ID.

- From the channel ID, we retrieves the channel descriptor from the array of channel descriptors (chann[]).

- Field head in the channel descriptor will then point to the list of channel values. Information in the channTable directs us to display the message value appropriately based on the message type and size.

Implementation of Trace Function

Trace function is implemented by labeling the traced variable, so that every time the traced variable is referenced during execution, the system state will be set to Break and the execution is suspended. There are two issues in the implementation: how to label traced variables and when variables are referenced during execution.

During execution, the code execution module (CEM) executes the vCode of the program and variables are referenced by their addresses in the memory pool. For

example, to execute the vCode LDVal (load the value of a variable), CEM refers to the variable by its address in memory pool (index of the memory array storageValue[]) and copies the value (storageValue[index].val.intValue) to the top of the working stack. Since every variable has a corresponding address in the memory pool and the address is referenced by CEM for execution, we decided to use the memory address as the label for the traced variables. The variable name and its memory address will form an entry in a global table, the traceTable, that is used to store all user specified traced variables. The memory address of a variable can be computed using the same approach as we described in the implementation of Show unction, referring to Figure 7. The data structures for the traceTable are shown in Figure 12:

When Trace function is called, user specifies the variable name and process id where the variable is in. CEM computes the memory address of the variable and stores a new entry in the trceTable. When user wants to clear a traced variable, the clearTrace function is called and the corresponding entry of this variable is removed from the traceTable.

```
struct traceTabEntry      {
     char name[maxVarLength]; // variable name
     int  memLoc;             // index in mamory pool
};

traceTabEntry  traceTable[MaxNbrTraces];
```

Figure 12 : Data structures in C for Trace function

In the vCode set defined in CPSS, vCode instructions that have variable reference in their execution are listed in Table 1: vCode's that have variable reference. Whenever a vCode in Table 1 is executed, CEM will call a function checkVarTrace() to check if the variable referenced by this vCode is in the global traceTable. If yes, the system state will be set to Break and the execution will be suspended.

VCode Mnemonic	Description
LDVal	Load value of a variable onto stack
LDIndirect	Load indirectly of a variable onto stack
LoadBlock	Load a block of data to stack
Dereference	Replace pointer on the stack by value
STORE	Store value into memory location
SCANF	Get user input for a variable from stdin
FSCANF	Get user input for a variable from a file
PRINTF	Output to standard output
FPRINTF	Output to a file
LOCK	Lock a variable to prevent concurrent access
UNLOCK	Unlock a variable to release the access control
CopyBlock	Copy a block of data into another
CopyToNewBlock	Allocate a new block and copy data to it

Table 1: vCode's that have variable reference

4.1.5 Examining Status of Process

Functionality

Some debugging tools can only be applied to processes in certain execution states. For example, user can only use the Show function to examine variables in active processes, Step function can only be used for processes in Running state. So user needs to know the execution status of each parallel process before other tools are used. We developed a tool called Status to meet this requirement. When program execution is in suspension state, users can use the Status tool to specify a range of processes to show their execution statuses.

Data Structure and Implementation

Every application process is associated with a process control block (PCB) that stores various information needed for its execution. When a new process is created, a PCB is allocated for this process and appended to a global list of processes, which is implemented as a singly-linked list. To implement the Status function, we simply identify the useful information in the PCB of each process and display these information on per-process basis. The information contained in PCB is shown in Figure 13 : Data structure of Process Control Block in C as a data structure in C language.

```
typedef struct ProcDesStruct {
     int processID;       // process ID
     int PC;              // program count
     enum State state;         // process state
     enum Priority priority;        // scheduling priority
     struct ProcDesStruct *parent;
                          // pointer to parent's PCB
     int base;            // lowest address of process frame
     int B;     // lowest address of current function frame
     int T;               // current stack top
     int stackTopLim;
          // highest address of the current function frame
     int forallLevel;
          // forall nesting level of this process
     int numForallChildren;
          // number of forall children currently running
     float maxForallTermiTime;
          // most recent forall termination time
     int forallIdxAdr;    // memory address of forall index
     char repeatProcInGroup;
          // flag for implementing grouping option
     int forkCount;
          // number of fork children currently running
     float maxForkTermiTime;
          // most recent fork termination time
     int joinCount;
          // number of fork children terminated but not
          //matched with join
     float time;          // local clock of this process
     float wakeTime;
          // wakeup time if process state is Delayed
     int virProcessor;         // virtual processor ID
     int phyProcessor;         // physical processor ID
     int altPhyProcessor;
          // to save actual physical processor ID on
          //parameter evaluation
     enum ReadStatus readChannStatus;
          // status during channel read
     char accumSeqTime,
          // flag for accumulating sequential time

} ProcDescriptor, *ProcDexPtr;
```

Figure 13 : Data structure of Process Control Block in C

In the Status function, we display the following information for each process that is within the range of processes specified by user:

- Process ID
- Function name: the function name that the process is currently executing in. Using the same approach described in the implementation of Show function, we can get the function name in the global identifier table as identTable[funcIndex].name.
- Source line number: the source line number that the process is currently executing. We can get the source line number based on the PCB's process count (PC), which is equal to the vCode line number. vCode line number is associated with the source line number in the src2codTable as we described in section 4.1.1
- State: the current state of the process, the state field in PCB
- Virtual processor ID: the virProcessor field in PCB
- Physical processor ID: the phyProcessor field in PCB

4.1.6 Examining Memory Contents

Functionality

This function is used to display the memory contents for a specified range of address in the memory pool during program execution. It provides a debugging tool for the implementor of CPPE to monitor the process stacks during execution of the parallel program. Users of CPPE rarely need to use this function.

Data structure and implementation

Memory block can be allocated to different processes during parallel program execution. This function displays both the content of the memory currently used and the process ID that is currently using that memory location. With the data structure storageValue[] and storageOwner[] described section 4.1.4. which are two arrays parallel to each other, we can display the value of a specific memory location (storageValue[n].val.intValue in case of a integer data type or stroageValue[n].val.floatValue is case of a float data type) and the process ID that owns that memory location (stroageOwner[n]).

4.2 Performance Debugging Tools

In chapter 3, we have identified a set of important performance data and statistics for parallel program execution. We have also reviewed the important affecting factors for the performance of parallel programs in chapter 2. Based on these information, we designed and implemented a set of functions, known as performance debugging tools, to help the user to fine-tune their application program. Basically these tools are categorized into three main types: tool that sets time breakpoint, tools to set network architectures, and tools to report performance statistics. There are two major designing issues in CPSS that affect the design and implementation of the performance debugging tools. The first issue is how timing system is implemented in CPSS. The second issue is how parallel execution environment is simulated in CPSS. We will review these issues in the process of introduction of relevant tools.

4.2.1 Setting Time Breakpoint: Alarm

Functionality

Alarm function is used to suspend the program execution when a certain amount of time is reached. When the program execution is suspended when the alarm times out, user can investigate the program execution status, reset the network parameters, and gather performance data as an aid in debugging or performance analysis of a program. We provide a user interface to set the alarm time and a switch to either turn the alarm on or off. The alarm setting can be done before the execution of a user program starts or when the execution is suspended.

Timing System of CPSS

CPSS uses a simulated timing system rather than the real machine time. A global variable globalClock is used to simulate the global clock and is initialized to 0 when a new program execution starts. Every application process has its local clock and is initialized with the time on its parent's clock when the process is created. All performance statistics are based on this simulated timing system, and the design and implementation of our performance debugging tools also heavily depends on the mechanism how this timing system works.

The way the global clock advances during program execution is decided by the Time Slicing mechanism employed by CPSS. Parallel execution of application processes in CPSS is simulated by time slicing. The execution of a parallel program is divided into quanta, each quantum lasting q clock cycles (or time units) where $q > 0$. During program

execution, each physical processor is assigned a quanta in round-robin manner. Each physical processor maintains a circularly linked list of processes running on it. These processes use the quantum allocated to this physical processor in round-robin fashion. During each quantum, the scheduler traverses the list of physical processors, and schedules a process to run from the circularly linked list of processes on a physical processor in a round-robin fashion. If the scheduled process is able to run, it executes until the time slice of q time units expires or it is put to sleep by some event. The scheduler then gets the next physical processor in the list of physical processors and schedules a process to run on the second physical processor. When the last processor in the list of physical processors finishes its time slice, the global clock (globalClock) is advanced by q time units to the next quantum and a new quantum begins. Such a quantum simulates q time units of parallel execution on all physical processors on a real parallel machine.

In addition to the global clock, each process has its local clock (field time in PCB). This local clock is needed for the process to synchronize with the global clock and with other processes. Every time the process finishes executing a vCode instruction, its local clock is incremented by the cycle count of that instruction, which is defined based on the complexity of the instruction. When the local clock reaches or exceeds the time on the global clock, the process knows that its time slice in this quantum is up. The scheduler will then schedule a process on another physical processor to run.

Implementation of Alarm Function

Alarm is set by assigning a float value to a global variable alarmTime that records the alarm time. When program execution proceeds, before CEM executes a vCode instruction, it checks the global clock time against the alarm time. When the global clock time reaches or exceeds the alarm time, system state will be set to Break and this cause the program execution to be suspended. We can use a global flag as an alarm switch. CEM only checks the alarm time when the alarm switch is turned on.

4.2.2 Setting Network Architectures

Functionality

When CPPE is started, CPSS sets up a default wormhole-routed network. The network parameters in CPSS include network topology, dimension and size, number of virtual channels between two adjacent nodes, buffer size of a virtual channel, size of a packet (a packet is the basic unit carrying the address of the destination node for routing purpose), size of a flit (flit is the smallest unit of information for transmission), size of the header in a packet, size of the data in a packet, message startup delay and network link delay. All these parameters are configurable in CPSS. We designed a set of tools into CPSS so that the user can examine the current configuration of the network and re-configure the network by redefining some or all of the parameters before starting running the application program.

The Simulated Parallel Computer Network and Data Structures

CPSS uses an uni-processor to simulate a parallel execution environment. The network module has a set of data structures that are used to simulate the parallel environment and

support the virtual-to-physical architecture mapping. It functions under the control of the network manager, which is responsible for allocating network resources, message routing, detecting and resolving deadlock if any. The network also uses the global clock mentioned before. In each quantum, all active packets that are not blocked are advanced by one link. In addition, the network manager provides a well-defined network API which makes the network reconfiguration easy. The following are the major configurable data structures of the network API which will be used in the designing and implementation of performance debugging tools:

- Network topology: the network architecture is defined by three global variables: phyTopo (network type), phyTopoDim (network dimension), and phyDimSizes[] (size of each dimension). Network manager uses these variables in message routing and communication cost computation.

- Network type: currently CPSS can simulate both wormhole-routed network and packet switching network. The network type is defined by a global variable called networkType and it tells the network manager what type of network it should simulate in message routing.

- Network communication parameters: for wormhole-routed network, all communication parameters are defined in a global variable of structure para. Network manager uses parameters in this structure to navigate the messages and calculate the communication cost.

- Timing system: the network module uses the same global clock as we describe before. It also defines a Basic Communication Delay, which is the basic time to communicate a message packet between two processors with a direct physical communication link.

Implementation of Network Configuration Tools

From the data structures of network API described above, we identified the following network parameters which may affect the program performance and implemented corresponding tools to enable the user to re-configure the corresponding parameters. The rationale for requiring these tools are also described.

- Network topology: network topology affects the connectivity between network nodes, which in turn affects the distance between nodes and the traffic contention in the network. To study the effect of network topology on the performance, we implemented the tool for changing network topology. This can be done by simply providing users with the interface to change the value of the three global variables that define a network topology: phyTopo (for network type), phyTopoDim (for network dimension) and phyDimSizes[] (for size of each dimension).

- Network type: because of the existence of different routing techniques used in the world of parallel computer, CPSS supports multiple network types to accommodate the user applications, at the same time, provides the performance statistics to evaluate the pros and cons of different network types for different types of applications. We implemented a tool to allow users to specify the network type

before execution of the application program or change network type during program execution. This can be done by providing users with the interface to change the value of the global variable networkType.

- Number of virtual channels per link: a virtual channel is a logical link between two adjacent nodes. Multiple virtual channels can be multiplexed on one physical link. This parameter affects the communication performance in a way that having more virtual channels enhances network throughput but increases the routing latency. This parameter is defined as a field in the global variable of structure para as nbrLanesPerLink. To study the effect of this parameter on the performance, we implemented a Display and Change function by providing a user interface to display or change the value of para->nbrLanesPerLink.

- Flit size: in wormhole-routed networks, flit is the smallest unit of information for transmission. Small flit size helps to reduce network latency. However, If the flit size is too small, flit overheads may overweigh the benefits of wormhole routing [13]. Other factors deciding the choice of flit size include network size, routing scheme, link bandwidth, and router design [14]. This is a parameter defined as a field in the global variable of structure para as flitSize_B. To study the effect of this parameter on the performance, we implemented a Display and Change function by providing a user interface to display or change the value of field flitSize_B of structure para.

- Packet size: a packet is the basic unit carrying the address of the destination node for routing purpose. Factors influencing the choice of packet size include the routing

scheme, link bandwidth, router design, and network traffic intensity [14]. This is a parameter defined as a field in the global variable of structure para as packSize_B. To study the effect of this parameter on the performance, we implemented a Display and Change function by providing a user interface to display or change the value of field packSize_B of structure para.

- Virtual channel buffer size: each virtual channel is associated with a buffer. Larger buffer size may improve network performance [13][15], but buffer size equal to packet size will effectively reduce the benefit of wormhole routing to that of packet switching. Increasing buffer size can also help to support large networks when the flit length is not long enough to carry node addresses. This is a parameter defined as a field in the global variable of structure para as buffSize_F. To study the effect of this parameter on the performance, we implemented a Display and Change function by providing a user interface to display or change the value of field buffSize_F of structure para.

- Message initialization cost: the startup cost of a message is primarily due to message and packet initialization overheads and buffer management. A message is first divided into packets which are initialized with the destination address, the sequence number and other routing information. Every packet is then buffered until the network port is available, and the packet is injected into the network. There are two parameters defined in para: msgInitCost and packInitCost, which stand for message initialization cost and packet initialization cost respectively. To study the effect of the message startup cost on the performance, we implemented a Display

and Change function by providing a user interface to display or change the value of field msgInitCost or packInitCost of structure para.

- Header and other flit speed ratio: the ratio between header flit speed and data flit speed. The latency for a flit to move from one node to the next node on the path differs between a header flit and a data flit, because in addition to the normal latency for a data flit, header flits incur router decision time and virtual channel allocation time. The ratio between the header flilt speed and data flit speed affects the overall network performance. This is a parameter defined as a field in the global variable structure para as headOtherFlitSpeedRatio. To study the effect of this parameter on the performance, we implemented a Display and Change function by providing a user interface to display or change the value of field headOtherFlitSpeedRatio of structure para.

- Basic communication delay: this is the basic time to communicate a message flit (or packet) between two processors with a direct physical communication link. The effect of the basic communication delay on performance varies between a computation-bound program and a communication-bound program. By saying a computation-bound program, we mean a program which spends much more time on local computation than inter-processor communication. On the contrary, a communication-bound program spends a significant amount of time in inter-processor communication. CPSS defines a global variable linkDelay to be used in communication cost evaluation. To study the effect of the basic communication

delay on the performance, we implemented a Change function by providing a user interface to change the value of linkDelay.

- Communication congestion: in packet switching network, basic communication delay is adequate to reflect the communication delay when the message traffic is low enough that there is no interference between messages that might result in congestion delay. Some programs have more frequent communication that travels longer paths in the network, resulting in the potential of message congestion and communication delay. CPSS defines a global variable congestionOn to signal the calculation of the communication delay. We implemented a function so that user can have the option to either turn the congestion on or off for two reasons. First, it gives the opportunity to determine if there is any performance degradation resulting from congestion in the communication network. Second, the detailed simulation of message flow in the network is very time consuming, and therefore may significantly increase the program simulation time. For programs that do not suffer from congestion problems, the user can turn this option off to make the simulation run more quickly.

4.2.3 Varying Processor Speed

The simulator can reproduce multiple executions of a non-deterministic application. This is particularly useful for applications whose behavior is considerably different from one run to another depending on the output of race conditions.

In CPPE, multiple executions are implemented by varying the relative processor speed. Race conditions can be created by the 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. If the two messages compete for the same physical link, the sending order would decide which message will get the link first.

At run time, users will be asked to provide an integer Random Number Seed, which will be used to create a random number r_i between 0 and 1 for each physical processor i that will be used to increase the speed by a factor of $1/r_i$. This randomly selected speed factor for each processor will remain in effect throughout the subsequent program execution.

4.2.4 Program Mapping

Functionality

In CPPE parallel programmers usually write an application using an architecture most natural and efficient to program performance. This is called virtual architecture programming, with the architecture referred to as virtual architecture and its processors as virtual processors. Virtual architecture programming improves the programmability of message-passing applications, however, the topology and size of the physical machine may not match that of the virtual architecture. Processors that constitute the physical machine are physical processors. At run time, virtual architecture will be mapped to the available physical architecture, and the performance of a parallel program is really calculated based on the physical architecture. The objectives of mapping are to minimize communication cost among communicating processes, and to balance the workload among physical processors. The performance debugger provides a library of different types of mapping, which currently supports the Default mapping, Identity mapping,

Random mapping, Ring-to-Line mapping, Torus-to-Mesh mapping and User-defined mapping.

Data Structures

A global variable curMapping is used for the user to specify the desired mapping type, and a virtual-to-physical mapping table, virPhyMapTab, which is implemented as an array, is use to store the data that defines the specific mapping function. In CPSS, simulation of program execution utilizes absolute IDs for processors so that simulation routines are generic and can be used for all types of topology. In the mapping table, we use the index of virPhyMapTab as the virtual processor ID, and the indexed value of virPhyMapTab as the physical processor ID.

Implementation

There are two levels of program mapping in CPSS. The first level is the mapping from processes to virtual processors. The second level is the mapping from virtual processors to physical processors.

- Process-to-virtual-architecture mapping: this is usually accomplished in the application program. Often in the application program the user specifies the ID of the virtual processor on which a process will run.

  If the user does not provide a virtual processor for a new process, at run time the process is mapped directly to a physical processor, bypassing the virtual processor level. The physical processor allocated to the new process is determined by a default

processor allocation algorithm which is implemented in CPSS. The default allocation criterion is to balance the work load among existing physical processors.

- Virtual-to-physical-architecture mapping: in CPSS, simulation of program execution utilizes absolute IDs for processors, so the mapping is accomplished by mapping the virtual processor ID to a physical processor ID. A virtual-to-physical mapping table (virPhyMapTab) is used to store the mapping function. with the index of virPhyMapTab being the virtual processor ID, and the indexed value of virPhyMapTab being the physical processor ID. The physical processor ID will be used to match the actual physical processor in the physical processor table (phyProsorTable). See Figure 14.

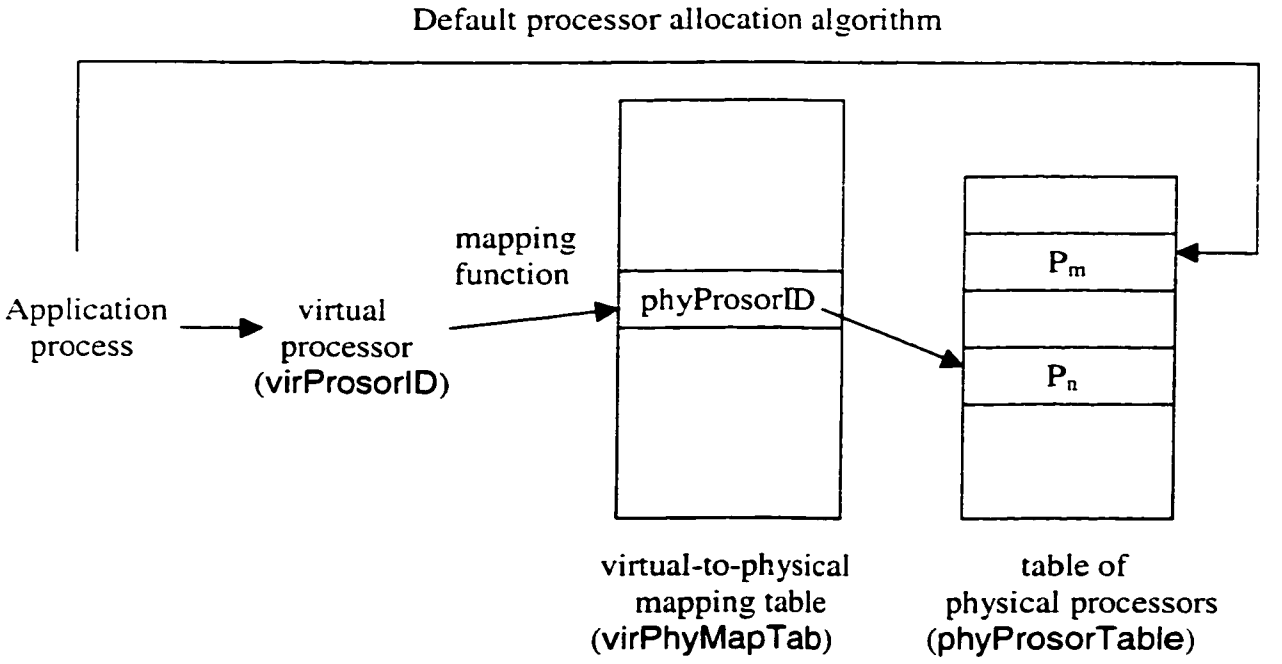


Figure 14 : Mapping of a process to a virtual processor then to a physical processor

Effectively, a process-to-physical-processor table can be derived from the process to virtual processor mapping and the virtual processor to physical processor mapping so that each parallel process will be allocated to run on an appropriate physical processor based on this table. Table 2 shows a sample mapping from parallel processes to physical processors. In this example, every two successive parallel processes are linearly mapped to one physical processor.

Parallel process ID	1	2	3	4	5	6	7	8
Virtual processor ID	1	8	2	7	3	6	4	5

Process to virtual processor mapping

Virtual processor ID	1	2	3	4	5	6	7	8
Physical processor ID	1	1	2	2	3	3	4	4

Virtual processor to physical processor mapping

Parallel process ID	1	2	3	4	5	6	7	8
Physical processor ID	1	4	1	4	2	3	2	3

Process to physical processor mapping

Table 2: Process-to-physical-processor mapping

The performance debugger provides a library of different types of virtual-to-physical mapping. The design strategy of mapping functions is to maximize the resemblance between a virtual architecture and an available physical architecture in terms of their communication behavior, so that the advantage of a particular virtual architecture for an application program, such as the minimized communication cost between

communicating processes and balanced work load between processors, can be carried over to the available physical architecture. Currently the library supports six types of mapping with algorithm for each type of mapping described as following.

1. Identity Mapping

A virtual processor ID is mapped to a physical processor with the same ID. This can happen only when the total number of physical processors is equal to or more than the total number of virtual processors. The algorithm in C language is shown in Figure 15.

```
int virAbsId;        // virtual processor absolute ID
int nbrVirProsors;   // total number of virtual processors
int nbrPhyProsors;   // total number of physical processors

if (nbrVirProsors <= nbrPhyProsors)
   for (virAbsId = 0; virAbsId < nbrVirProsors; virAbsId++)
     virPhyMapTab[virAbsId] = virAbsId;
```

Figure 15 : Algorithm for Identity mapping in C

2. Random Mapping

Random mapping is simulated by mapping a virtual processor ID number to a physical processor ID number that is randomly produced by the host machine. We can first make an identity mapping as described above. Then we swap the entries in the mapping table randomly using the system function rand(). Random mapping is also supported only when the number of physical processors is equal or greater than the number of virtual processors. The algorithm in C language is shown in Figure 16.

```
if (nbrVirProsors <= nbrPhyProsors) {
  for (virAbsId = 0; virAbsId < nbrVirProsors; virAbsId++)
    virPhyMapTab[virAbsId] = virAbsId;
  for (virAbsId = nbrVirProsors-1; virAbsId > 2;
       virAbsId--)
    swap(virPhyMapTab[virAbsId],
      virPhyMapTab[(rand() % (virAbsId-1)) + 1]);
}
```

Figure 16 : Algorithm for Random mapping in C

3. Default Mapping

In default mapping, if the number of the virtual processors is not greater than the number of physical processors, the mapping uses the same algorithm as the identity mapping. Otherwise, virtual processors are linearly divided into blocks, with the block number equals to the number of physical processors. Each block of virtual processors is mapped to the corresponding physical processors. If the number of virtual processors is a multiple of the number of physical processors, each physical processor gets the same amount of virtual processors mapped to itself. Otherwise, the first several physical processors get one more virtual processor, while the other physical processors get the default amount of virtual processors. The algorithm in C language is shown in Figure 17.

```
// How many virtual processors are mapped to the same
// physical processor
blockSize = nbrVirProsors / nbrPhyProsors;
// how many virtual processors are left over
remains = nbrVirProsors % nbrPhyProsors;
if (nbrVirProsors <= nbrPhyProsors) // Identity Mapping
    for (virAbsId = 0; virAbsId < nbrVirProsors; virAbsId++)
       virPhyMapTab[virAbsId] = virAbsId;
else if (remains == 0) {
   // Linear Group Mapping
   virProsorNbr = 0;
   for (phyProsorNbr=0; phyProsorNbr<nbrPhyProsors;
                                        phyProsorNbr++)
      for (i = 0; i < blockSize; i++)
         virPhyMapTab[virProsorNbr++] = phyProsorNbr;
}
else { // remains != 0  && remains < nbrPhyProsors
    // Linear Group Mapping
    virProsorNbr = 0;
    blockSize += 1;
       // first physical processors get one more virtual
       // processor
    for (phyProsorNbr = 0; phyProsorNbr < remains;
                                        phyProsorNbr++)
       for (i = 0; i < blockSize; i++)
          virPhyMapTab[virProsorNbr++] = phyProsorNbr;
    blockSize -= 1;
       // others get the default number of virtual processors
    for (; phyProsorNbr < nbrPhyProsors; phyProsorNbr++)
       for (i = 0; i < blockSize; i++)
          virPhyMapTab[virProsorNbr++] = phyProsorNbr;
}
```

Figure 17 : Algorithm for Default mapping in C

4. Ring-to-Line Mapping

This type of mapping is to map a virtual architecture of Ring to a physical architecture of Line, and it is supported when the number of virtual processors equals to the number of physical processors. For a Ring architecture, the maximum distance between two

processors in terms of hops is half the total number of processors n. While for a Line architecture, the distance between processors linearly increases with the process number. We developed a jump() function to map the virtual processor to the physical processor in a way that distance between any two neighboring virtual processors is at most 2. The algorithm is shown in Figure 18:

```
int jump(int x, int n) {
     if (x<=((n-1)/2))
          return 2*x;
     else
          return 2*(n-x)-1;
}

void setRingMap() {
     int i;
     for (i = 0; i<nbrVirProsors; i++)
          virPhyMapTab[i] = jump(i, nbrPhyProsors);
}
```

Figure 18 : Algorithm for Ring-to-Line Mapping in C

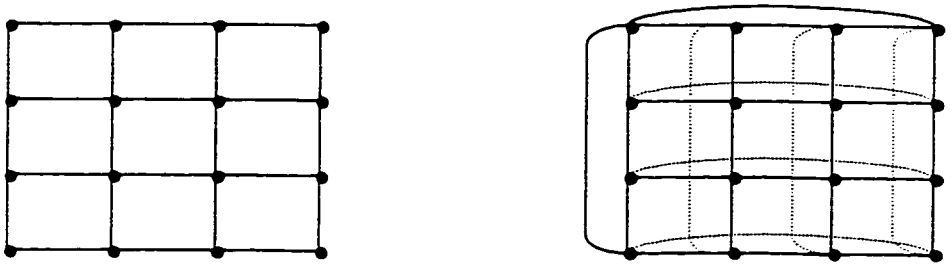
As an example for total number of virtual processor of 8, the effective result of Ring-to-Ling mapping is shown in Table 3:

Virtual Processor ID	0	1	2	3	4	5	6	7
Physical processor ID	0	2	4	6	7	5	3	1

Table 3 : Ring-to-Line mapping

5. 2D Torus-to-Mesh Mapping

This type of mapping is to map a virtual architecture of 2D Torus to a physical architecture of 2D Mesh, and it is supported only when the number of virtual processors equals to the number of physical processors in both dimensions. Figure 19 shows a sample two-dimensional Torus and Mesh topology:



2D Mesh (4 x 4)

2D Torus (4 x 4)

Figure 19 : Mesh and Torus topology

A Torus topology can be viewed as connecting the two ends of each dimension of a Mesh topology, just as a Ring topology can be viewed as connecting two ends of a Line topology. Therefore the basic algorithm for the Ring-to-Line mapping is also applied to the Torus-to-Mesh mapping, except that in Torus-to-Mesh mapping, Cartesian ID is used to describe the relative position in a multi-dimensional topology.

We first transform the virtual absolute processor ID (virAbsID) into virtual Cartesian ID (virCartID) by calling the function cart_ID(). In function cart_ID() the "row-major ordering" algorithm is used to transform an integer absolute ID into a Cartesian ID which is a multi-dimensional Cartesian coordinator. Then for each dimension i of the virtual Cartesian ID (virCartID[i]), we use the same jump() function as in the Ring-to-Line mapping to map it to a physical Cartesian ID (phyCartID[i]). Finally

the physical Cartesian ID is transformed to a physical absolute processor ID (phyAbsID) by calling function abs_id() that also uses the "row-major ordering" algorithm to transform a Cartesian coordinator to an integer absolute ID. The detailed algorithm is shown in Figure 20:

```
void cart_ID( enum ArchType topology, int topoDim,
              int dimSizes[], int abs_id, int cart_id[] ) {
   int i;
   switch (topology) {
      case TORUS:
         for (i = topoDim; i > 0; i--) {
            cart_id[i] = abs_id % dimSizes[i];
            abs_id /= dimSizes[i];
         }
         break;
      case OTHER TOPOLOGY
         … …
      default:
         printf("cart_ID: invalid topology\n");
         exit(-1);
   }
} // cart_ID

int abs_ID( enum ArchType topology, int topoDim,
            int dimSizes[], int cart_id[] ) {
   int i, abs_id;
   abs_id = cart_id[1]; // first coordinate
   for (i = 2; i <= topoDim; i++)
      abs_id = abs_id * dimSizes[i] + cart_id[i];
   return abs_id;
} // abs_ID

void setTorusMap() {
     int virAbsID, i;
     for (virAbsID = 0; virAbsID < nbrVirProsors;
          virAbsID++) {
        cart_ID(virTopo, virTopoDim, virDimSizes, virAbsID,
                virCartId);
        for (i=1; i<=virTopoDim; i++)
           phyCartId[i] = jump(virCartId[i],
                               phyDimSizes[i]);
        virPhyMapTab[virAbsID] = abs_ID(phyTopo,
           phyTopoDim, phyDimSizes, phyCartId);
     }
}
```

Figure 20 :Algorithm for Torus-to-Mesh mapping in C

6. User-defined Mapping

This function enables user to declare virtual-to-physical mapping after loading the vCode file and before each execution. User specify a custom mapping function interactively by entering a mapping function string via the debugger's dialog box, which is processed by **lex** and **yacc** facilities to map the virtual processors to physical processors.

As an example, a one-to-one mapping from a two-dimensional torus to a two-dimensional mesh can be translated as following:

```
torus2mesh(x, y) = [x<=(#1-1)/2 ? 2*x : 2*(#1-x)-1,
                    y<=(#2-1)/2 ? 2*y : 2*(#2-y)-1].
```

where *#n* denotes the size of the n^{th} dimension of the virtual architecture, and the pair (*x*, *y*) denotes the Cartesian coordinate of the virtual processor in the virtual architecture. The square brackets enclose the right-hand side of the mapping function. The pair of values inside the square brackets denotes the Cartesian coordinate of the physical processor in the physical architecture. The final dot is used to signal the end of the mapping string so that users can break the map string in several lines during inputting.

A many–to-one mapping from a two-dimensional torus to a two-dimensional mesh can be expressed as follwoing:

```
torus2mesh(x,y)=
     [x<=(#1-1)/2 ? (2*x)/(#1/$1):(2*(#1-x)-1)/(#1/$1),
     y<=(#2-1)/2 ? (2*y)/(#2/$2):(2*(#2-y)-1)/(#2/$2)].
```

where *Sn* denotes the size of the n^{th} dimension in the physical architecture. The denominator (*#i/Si*) in the physical Cartesian coordinate provides the effect of block cyclic mapping.

In general, a mapping function string has the following format:

```
mapfunction(x1, x2, ..., xm) = [y1, y2, ..., yn].
```

where m is the dimensionality of the virtual architecture. (x_1, x_2, ... x_m) is the Cartesian coordinate of the virtual processor in the virtual architecture. n is the dimensionality of the physical architecture, (y_1, y_2, ... y_n) is the Cartesian coordinate of the physical processor in the physical architecture, and y_i is a function of the virtual Cartesian coordinate and the sizes of each dimension in the virtual and physical architectures. In other words, y_i can be expressed as following:

```
yi = fi(x1,x2,...,xm, #1, #2, ..., #m, $1, $2, ..., $n)
```

All basic mathematical operations and relations in C are supported. Conditional statement can be expressed using the ? : operator.

4.2.5 Performance Statistics

Functionality

The ultimate goal of parallel computing is to reduce the total execution time, which can be evaluated by the speedup by parallel computing compared with sequential computing. Due to the complexity of parallel computing, parallel programmers usually need more information about the execution process in addition to the final result of

speedup in order to better understand the behavior of the program. We identified some important performance statistics and implemented corresponding tools for users to get these data.

Basically we defined two types of performance statistics in CPSS. The first type is the run-time data which can be examined at any execution suspension point. These data can provide information about the program performance in a specific aspect and within a specific period of time during execution, such as the parallel speedup since last breakpoint and the usage of a specific processor since last profile. This helps the user to monitor the performance during various phases of the program execution. Users can concentrate on the performance of localized segments of the parallel programs. For example, it can be used to remove the effects of data initialization from the overall performance statistics. For the run-time data, we designed three tools to be used when program execution is suspended: Time, Utilization and Profile.

The second type is the final performance statistics provided at the end of program execution. These data provide the overall performance statistics of the program, such as the overall speedup of the program by parallel execution. For the second type, the final performance statistics are provided at the end of program execution following the program output.

Data Structures

All performance data are based on the timing system described before. When CPSS executes the vCode instruction, it uses an estimated execution time for each vCode instruction and keeps a running total of the execution time of the program. The number of

time units charged for each vCode instruction varies depending on the relative complexity of the instruction. We implemented into CPSS the following data structures that will be used to produce the performance statistics:

- globalClock and seqTime: the running total of the execution time of the program distinguishes between parallel execution time and sequential execution time. The global variable globalClock in CPSS we described before stands for the parallel execution time, because it increments only after all physical processors have been scheduled to run for a time quantum. CPSS uses another global variable seqTime to record the total execution time if the program is executed in a sequential mode. Each parallel process has a flag in its PCB to indicate whether the current vCode instruction is parallel-execution specific or not. seqTime increments as long as the current vCode instruction is not parallel-execution specific.

- prevParaTime and prevSeqTime: when program execution resumes from a breakpoint, we record the current globalClock time and seqTime as the prevParaTime and prevSeqTime respectively. This allows us to get the performance statistics within the time window of two consecutive breakpoints.

- nbrUsedProcessors: this stands for the number of actually used physical processors. In CPPE, users can specify the process-to-virtual-processor mapping and virtual-to-physical-architecture mapping. If user does not specify such mapping, at run time, the application processes will be mapped to physical processors by a *default processor allocation algorithm*. The default allocation criteria is to balance the work

load among existing physical processors. In either case, the actual number of physical processors used will be recorded by the global variable nbrUsedProcessors.

- phyProsorTable: information required to simulate a physical processor is stored in a structure called physical processor descriptor. phyProsorTable is used to store the physical processor descriptors of all available processors in the parallel computer system. Some information we stored in the physical processor descriptor that are related to performance evaluation include virTime, the virtual processor running time since the beginning; profileTime, the virtual processor running time since last profile; virTimeLastBreak, the virtual processor running time since the last breakpoint. These information reflect the usage of a particular physical processor for a certain period of time, which will be used in the implementation of the profile function.

Implementation of Time Function

Time function is used to examine the run time performance when the program execution is suspended. User can first set two source lines as breakpoints and then examine the performance between these two source lines. This is useful for focusing attention on the performance of localized segments of the program. In this function, we display the following information to the user upon invoking the Time function, using the values of the variables we described above:

- Elapsed time since the beginning: this is the time of the variable globalClock, which is the elapsed parallel time since the beginning of the program.

- Elapsed time since last breakpoint: this is the elapsed parallel time since the last breakpoint. The value is (globalClock – prevParaTime).

- Parallel speedup since the beginning: the parallel speedup is reflected by the ratio between execution time in sequential mode and the execution time in parallel mode (SequentialTime/ParallelTime). The speedup since the beginning can be calculated by seqTime/globalClock. This is an approximation of the theoretical speedup.

- Parallel speedup since last breakpoint: this is the ratio between the elapsed sequential time (seqTime – prevSeqTime) since the last breakpoint and the elapsed parallel time since last breakpoint (globalClock – prevParaTime).

- Number of processors used: the number of physical processors used up to this breakpoint is stored in the variable nbrUsedProcessors.

Implementation of Utilization Function

Utilization is used to measure how efficiently a program uses a particular parallel architecture. The measurement is based on the utilization of each physical processor defined as the proportion of the time the processor is actually running. User can use this function when the program execution is suspended to get a table of utilization information of the specified range of processors up to that point in the execution. The utilization information we displayed include:

- Utilization since the beginning: the actual running time of each processor since the beginning is recorded in the physical processor descriptor by the variable virTime.

When a process is scheduled to run, its PCB has a field phyProcessor to specify the physical processor ID the process is running on based on the user specified mappings or the default processor allocation algorithm described before. Whenever a process executes on this physical processor, this physical processor's virTime increments by the amount of the corresponding instruction cycles, see Figure 21. The utilization of a processor i is the percentage of its virTime against the globalClock (phyProsorTable[i].virTime/globalClock*100).

- Utilization since last breakpoint: similarly, the utilization of a processor i since last breakpoint is the percentage of its actual running time since last breakpoint (phyProsorTable[i].virTimeLastBreak) against the elapsed parallel time since last breakpoint (globalClock – prevParaTime). The variable virTimeLastBreak increments in the same way as virTime does (Figure 21), except that it is set to zero every time the program resumes execution from a breakpoint.

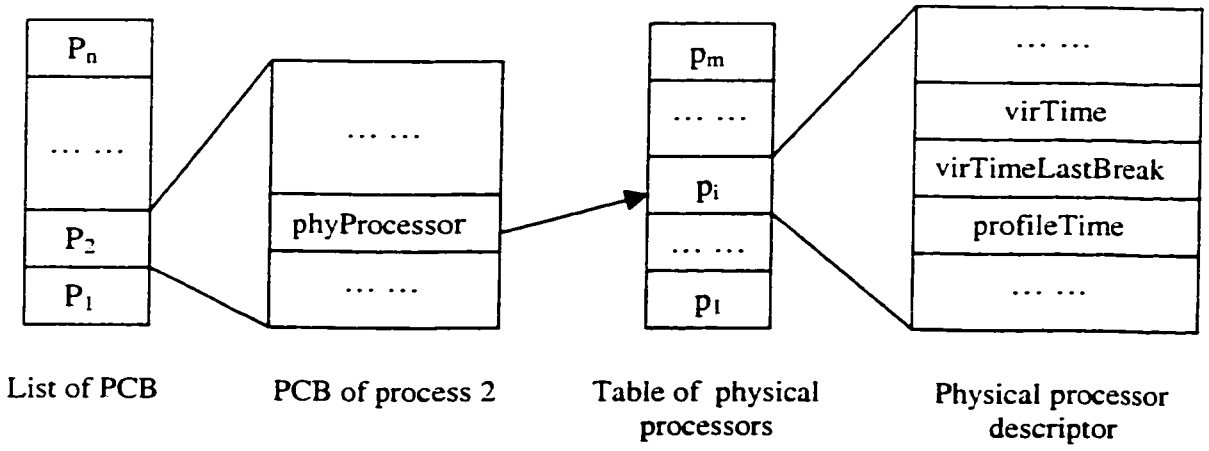


Figure 21 : Data structure related to Physical Processor Descriptor

We provide a user interface that allows the user to specify the range of processors displayed in this function.

Implementation of Profile Function

This is a function that creates a visual performance profile, that will help the user to understand the program performance at a glance. Figure 22 is a typical example of a performance profile, showing the physical processor utilization during successive time intervals of program execution.

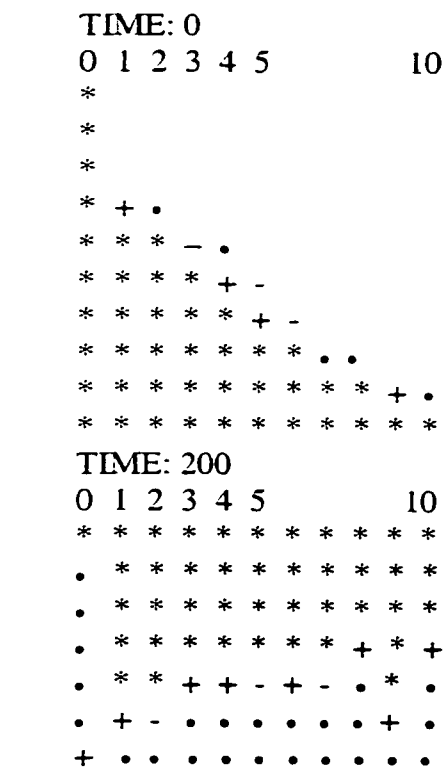


Figure 22 : Performance profile of a parallel program

The program starts at time 0. The physical processor numbers are given horizontally across the display. Time advances vertically down the display from top to bottom. In this case, each successive line of the display represents a time interval of duration of 20 time units. The marks indicate the processor utilization during each time interval: "*" indicates 75-100 percent utilization, "+" indicates 50-75 percent utilization, "-" indicates 25-50 percent utilization, and "•" indicates 0-25 percent utilization. To improve readability, a time stamp is inserted after certain amount of time (in this case 200) has elapsed.

Utilization of a processor i in the performance profile is the percentage of its actual running time (phyProsorTable[i].profileTime) during a user specified time interval in the profile (profileStep). The variable profileTime increments in the way as virTime does (Figure 21), except that it is set to zero every time after profile is reported for the latest time interval. Profile is generated from the function execute() in the Code Execution Module. execute() uses another timer profileReportTime to keep track of the next profile reporting time, which is the globalClock of the last profile reporting time plus the user specified time interval (globalClock + profileStep). When the globalClock exceeds the profileReportTime, the execute() generates a new line for the profile for the latest time interval.

We provide a user interface that allows the user to specify the time interval and the range of physical processors in the profile.

Overall Performance Statistics

Overall performance statistics is provided at the end of the program execution. It is the overall performance evaluation in the current execution environment, taking into account all performance affecting factors we reviewed in Chapter 2. The most important data that we concern is the speedup gained from parallel computing compared with the sequential computing. The other concern is the resource requirement, which is the number of physical processors used. In a simulation environment, user may also be interested in the actual time needed to run the application in the parallel computer simulator.

The overall speedup can be evaluated by the ratio between the execution time in sequential mode, which is the **seqTime**, and the execution time in parallel mode, which is the globalClock. The number of physical processors used is recorded in the variable nbrUsedProcessors. The actual execution time of the application program is the actual machine time on which the simulator is running on.

Chapter 5

Graphical User Interface

A graphical 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 design goal of the GUI is to make program developing and debugging easy for the parallel application programmers, at the same time the GUI can be easily configured to run in different environment and on different platforms.

5.1 Design Objectives

The most important objectives of the design of the CPPE GUI are as follows:

- Unified CPPE development environment: The major components in CPPE include CPC, CPCC and CPSS. CPC is the language for writing parallel application programs, while CPCC and CPSS are two independent components for compile and execution respectively. Plus the source code editing, user needs three separate executables in the whole developing process. The GUI design should accomplish a unified developing environment so that the user can configure and launch the CPPE

in one shot. All user activities involved in the developing process can be done through this GUI.

- User friendliness: Due to the complexity of parallel programming and multiple dimensionality of parallel performance, the GUI design should make the use of different debugging tools easy, and make the understanding and analysis of the program outputs and performance statistics easy.

- Easy configuration: Different environment and different user requirements can be satisfied by easy configuration without modifying the GUI source code. The GUI design should reduce or eliminate hard coding of environment setting and resource requirement, making adaptation as easy as possible.

- Multiple platform: CPPE is intended to be used in different teaching and researching environments with different platforms, most likely the UNIX workstation platform and the Windows PC platform. The GUI design should reduce the modification of the "engine" part of CPPE (CPCC and CPSS) and restrict the platform-specific code to the interface itself.

The stated objectives above are realized in the design and implementation of the GUI for both UNIX and Windows platforms.

5.2 Approaches to Realize the Design Objectives

5.2.1 Unified CPPE Development Environment

The GUI for both UNIX and Windows platforms share a common design objective and implementation strategy, which is to provide a unified parallel programming environment. CPPE needs three major functions to be a complete development environment: source file editing function, compile of the application programs written in CPC, simulation of parallel system and program execution. Considering that both UNIX and Windows platforms have powerful text editors coming with the operating system, we do not provide integrated text editing function in CPPE. Instead, user can use text editor outside CPPE in order to achieve best editing result. However, we provide user the option to either use the editor outside the CPPE environment or invoke the editor from within the CPPE interface. We can invoke an editor executable, such as emacs in the UNIX platform or notepad in the Windows platform, from an interface button or menu item and put it into background execution mode. In that case, the CPPE and the editor can run concurrently.

The compile function in CPPE is provided by CPCC. If used from command line, CPCC is an independent executable with its own command line input and console output. To incorporate CPCC into the unified CPPE environment, we need to modify its main function into a CPPE global function, which is called from an interface callback function. A generic output function is needed to replace the original output function to display the program output into a designated text widget in the CPPE GUI.

CPSS is the parallel system simulator that includes the functionality of network module, code execution module, and debugging monitor. When used from command line, CPSS has an interpret function that accepts the user command and invokes the corresponding functions from the network module, code execution module or debugging

monitor. When incorporated into the unified CPPE, the interpret function of CPSS will not be used. Instead, CPSS functions can be invoked from the callback functions of GUI components. The same generic output function can be used to display the program output into designated GUI components.

A main function is used to create the GUI. Functions of CPCC and CPSS are all registered with the callback functions of GUI components and invoked when corresponding GUI components are activated upon user events.

5.2.2 User Friendliness

The objective of user friendliness is realized by easy access of the different CPPE functionality from the GUI and the clear visualization of the program output, different debugging information and performance statistics for performance analysis.

User can access the CPPE function from main menu, option menus and function buttons. For the most frequently used functions, such as open a source file, compile a source file, run a parallel program, we provide the function buttons directly on the main frame. User can invoke a function by simply clicking the corresponding function button. In the case that user needs to select an item from several predefined options, such as selecting a physical parallel architecture, selecting a virtual-to-physical mapping pattern, selecting a source file from previously opened source files, we provide the option menus directly on the main frame. User can select an item by pulling and selecting from an option menu. All other functions are categorized and distributed in the GUI main menu.

For clear visualization of the program output, we use different windows for the different types of program display. The program output, debugging information and

performance data are displayed in the main output window of the GUI. Source code List function will open another window to display the source code. vCode List function also open another window to display the vCode. Help function will open a window to display the help information. All the windows are movable and resizable, so that user can easily check program execution status against the source code and vCode, with the Help window describing the functionality and significance of the debugging tools and program result. The program output, debugging information and performance data displayed in the GUI main output window can also be saved into a log file for further analysis.

5.2.3 Easy Configuration and Multi-Platform Support

CPPE configuration can be done through a configuration file, resource files and environment variables without changing source code and recompile.

CPPE is intended to be used on any host machine that has a C compiler. Compile conditional flags are used in the source code to adapt the simulator to different version of C compiler and the simulator can be built with or without GUI. Without GUI, CPPE runs in command-line mode with all debugging commands input from command line, which is more difficult to use but is faster. Currently, the simulator can work on UNIX workstations and PCs, with or without GUI.

5.3 GUI for UNIX Platform

5.3.1 Main Function

The GUI for UNIX workstation platform is developed using MOTIF-toolkit. A main function is developed to create the GUI main frame with widgets designated for different CPPE functionality. Each widget is associated with a callback function that in turn calls a user defined function in CPCC or CPSS. The program starts with the initialization of the GUI main frame. Then the main function enters an infinite loop to wait for the messages received from the GUI widgets. Based on the xlib (X window system) and Xt intrinsics (X toolkit library), MOTIF toolkit provides the functionality to create the loop, receive the messages from the GUI widgets, and invoke the callback functions of the widgets, that in turn invokes the corresponding CPCC and CPSS functionality.

Figure 23 illustrates the event-handling of a MOTIF application.

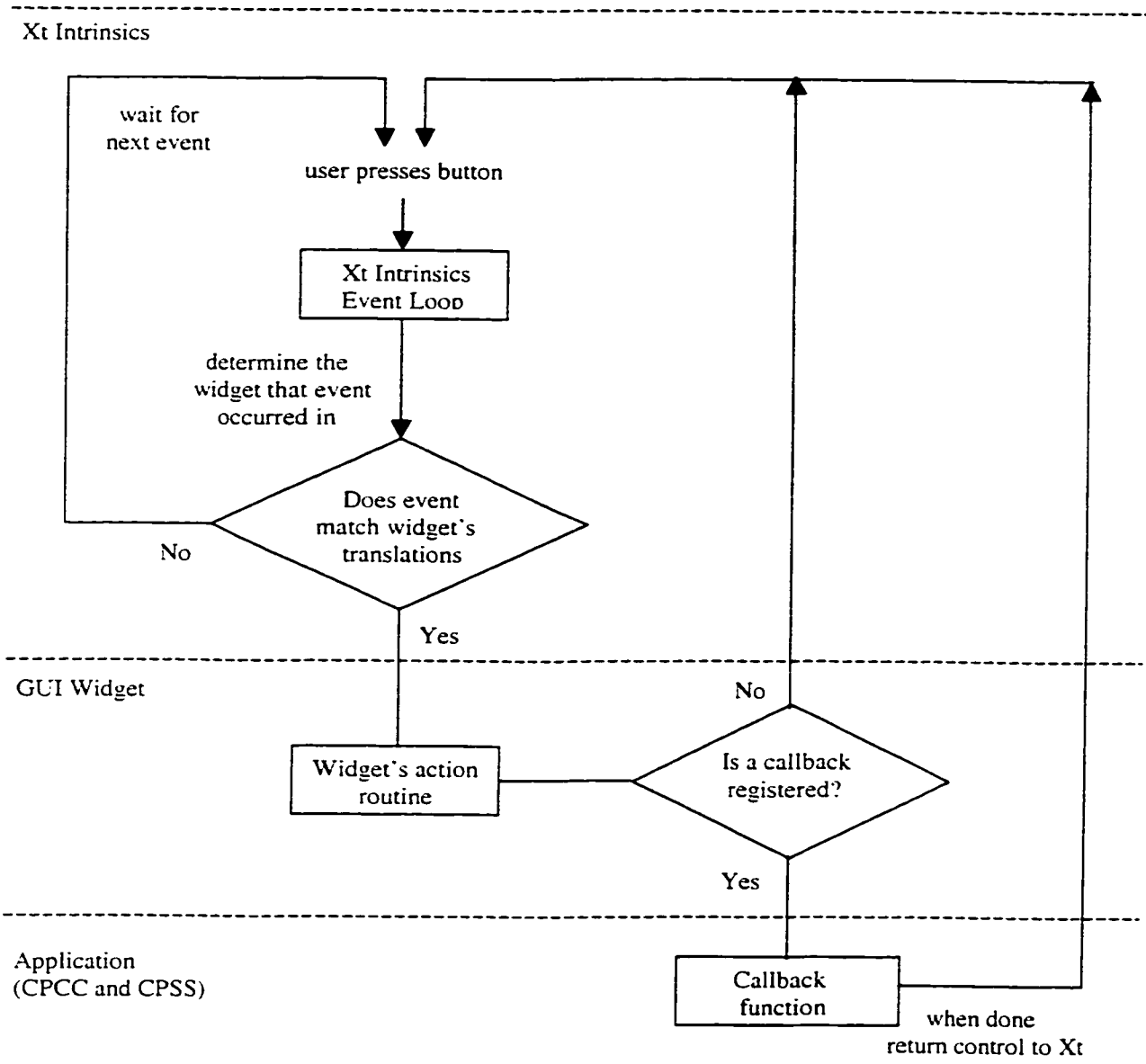


Figure 23 : Event handling of a MOTIF application

5.3.2 Main Frame

Figure 24 shows a typical layout of the CPPE GUI on the UNIX platform. The

window with the title *CPPE* is the main frame of the CPPE. The upper part of the main frame provides the main menu, function buttons and option menu for user to invoke the CPPE functions. The main output window in the main frame is the area that displays the application execution result, debugging information and performance statistics. The bottom of the main frame shows the current working path.

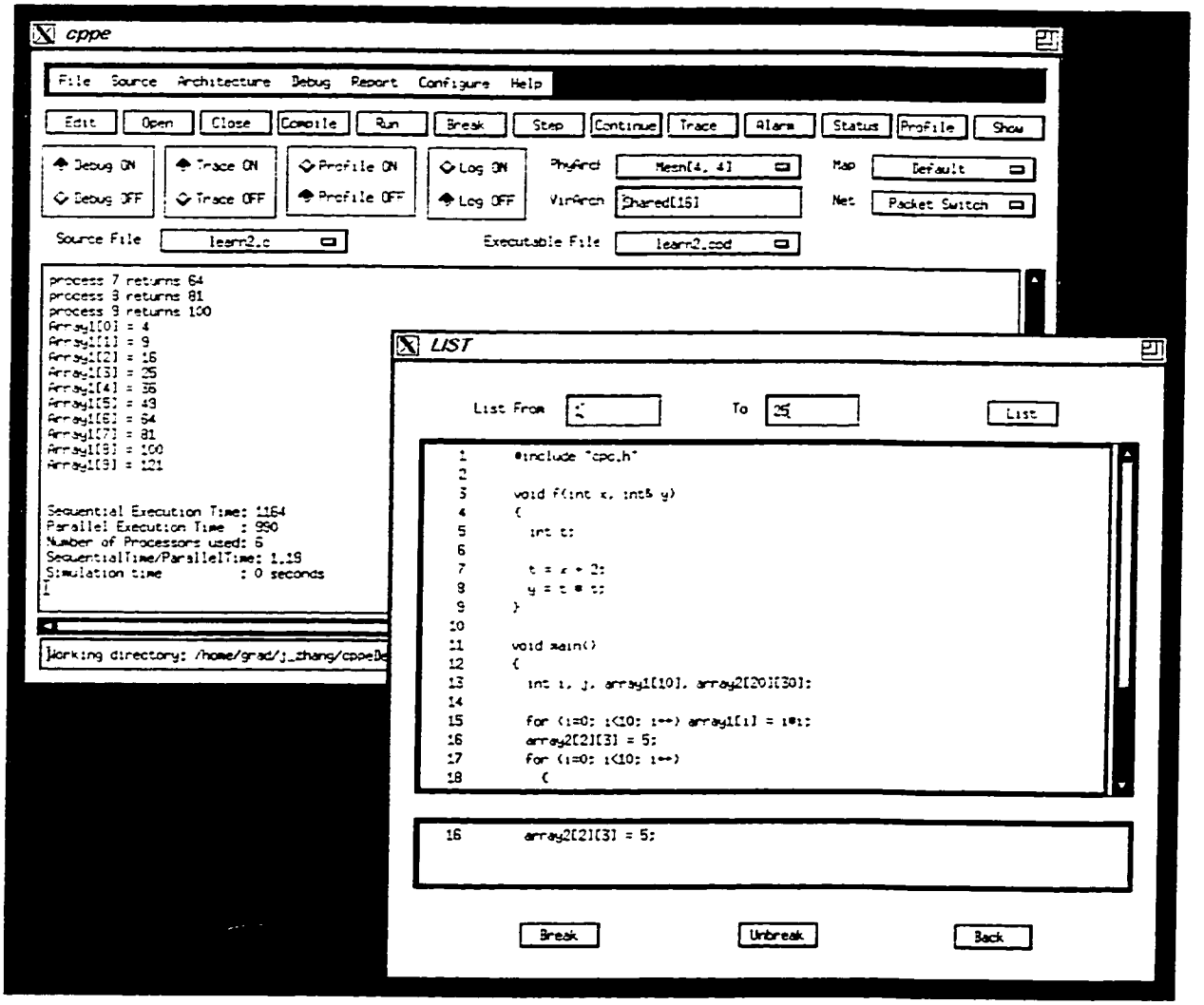


Figure 24 : CPPE GUI on UNIX workstation platform

The window with the title *LIST* is the source code listing window. User can specify the starting and ending line numbers of the source code to be displayed. This window can also be used to set source code breakpoints. There are two source code lists in this window. The list in the upper part of the window is used to display the source code with line number, called *source list*, and the list in the lower part of the window is used to display the source lines of the breakpoints, called *breakpoint list*. User can set a breakpoint by clicking on the desired source line in the *source list*, and this source line will be listed in the *breakpoint list*. To clear a breakpoint, user can click the source line in the *breakpoint list*, and this source line will be deleted from the *breakpoint list*.

5.3.3 Configuration

A resource file called CPPE is needed for the basic MOTIF resource specification for the GUI. A configuration file called cppe_motif.cfg is used to specify the basic features of the GUI, such as the default input path, the current physical architecture and network size, the current list of defined architectures, the current window size, etc. When CPPE is launched, a specific class in CPPE will try to find this configuration file and use it to initialize itself. An environment variable called XAPPLRESDIR is used to specify the path for the resource file CPPE. Another environment variable called CPPE is used to define the path to find the input .c and .h files in the compile and execution process.

5.3.4 Data Structures

The graphical user interface is an X application implemented as a widget hierarchy as shown in Figure 25.

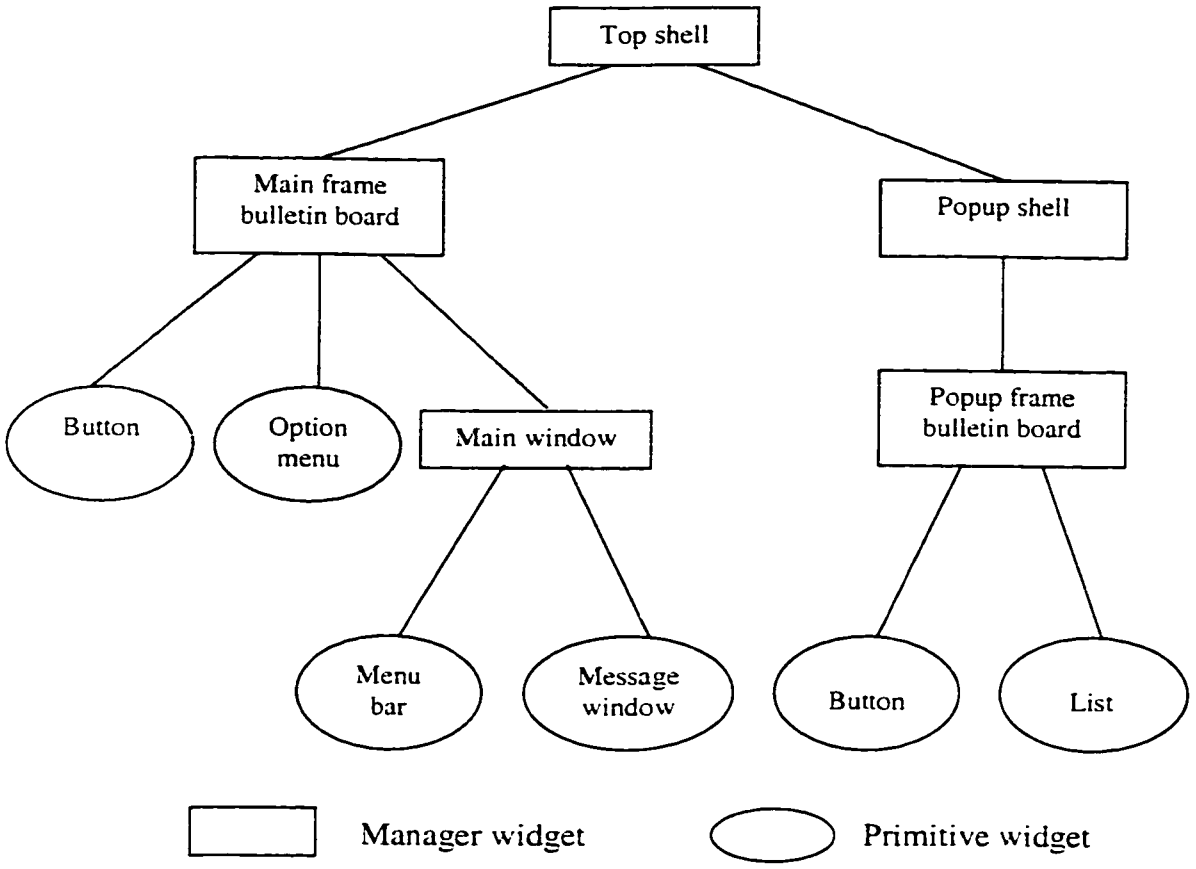


Figure 25 : Application widget hierarchy in CPPE GUI

Widget Hierarchy

The application first initializes the Xt Intrinsics, creates an application context to be used by the rest of the application, and creates a top shell widget in the widget hierarchy. The top shell widget contains an endless loop that waits for process events. Under the top shell, there are several manager widgets which are able to contain and manage their descendent widgets, which could be manager widgets or primitive widgets. Each primitive widget has a registered callback function which is invoked upon user

events. User events are received as Xt Intrinsics messages and are passed along the widget hierarchy towards the top shell until it is handled.

Main Frame Bulletin Board

This is the main frame of the CPPE GUI. On top of it we have built the main output window and the menus. The main output window is used to display program output and debugging information. The contents of the output window can be saved to a log file. To make the GUI user-friendly, we designed three kinds of menu: function button, option menu and main menu bar. Function buttons can be used to invoke those most frequently used functions in CPPE such as Open a source file, Compile and Execute. An option menu enables user to make a choice from some predefined options. The menu bar provides the comprehensive menu for CPPE functionality.

Popup Shell

Popup shell is initially created as a hidden shell. Unless it is explicitly popped up by the program, it will be invisible upon the start of the program. Popup shell has the ability of other manager widgets in terms of containing and managing descent widgets. In our program, we have three popup shells which are used for listing source code, listing vCode and displaying help information respectively.

Primitive Widgets

Primitive widgets are the graphical interface components that directly interact with user. Each primitive widget has a registered callback function which will be invoked upon user

events such as mouse clicking. The widget callback function serves as a container of CPCC or CPSS functions so that user can call CPCC or CPSS functions through the graphical interface. Callback function also updates the graphical interface accordingly based on the system execution status of CPPE.

5.4 GUI for Windows Platform

5.4.1 Main Frame

The GUI for the Windows PC platform is developed with MS-MFC. MS-MFC evolves with the Windows operating system so that it provides the most advanced graphical features in the GUI design. In addition, MFC provides an object-oriented development environment, with well defined graphic objects and hierarchical relations between graphic objects. This makes the application program easy to maintain and evolve in the future. With the evolution of MFC, we can continuously employ new features to enhance the graphical functionality of the CPPE.

Figure 26 shows a typical layout of the CPPE GUI on the Windows platform.

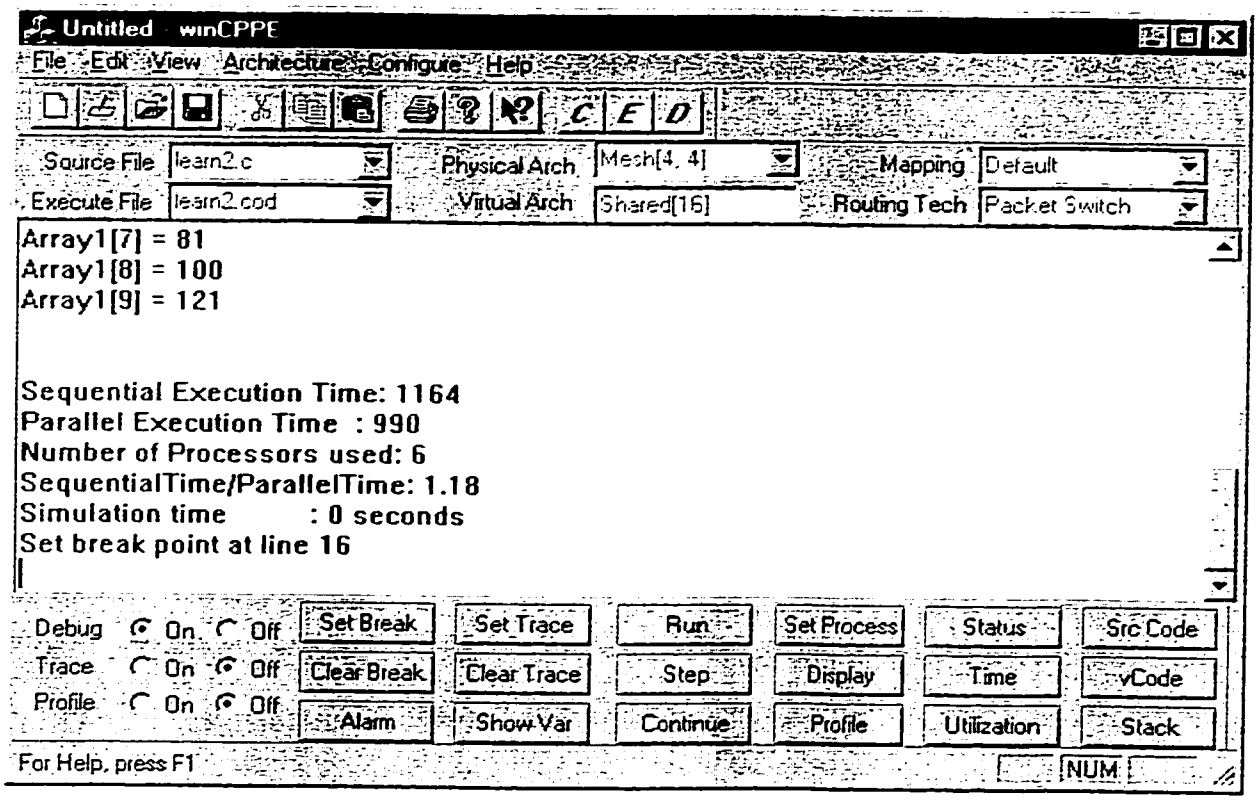


Figure 26 : Main frame of CPPE on Windows platform

5.4.2 Configuration

Same as the UNIX version, CPPE configuration can be done through configuration files and environment variables without changing source code and recompile. A configuration file called cppe_win.cfg is used to specify the basic features of the GUI. A specific class is developed to read this configuration file and define the GUI features. CPPE is used to define the path to find the input .c and .h files in the compile and execution process. On a PC platform, we can also define environment variables in the system's autoexec batch file. The following example shows how to define the environment variables CPPE in file autoexec.bat:

```
set CPPE=,;d:\cppe98\test
```

5.4.3 Data Structures

The CPPE GUI are constructed from a few application classes that are inherited from appropriate MFC base classes. Figure 27 shows the major application classes and their relationship.

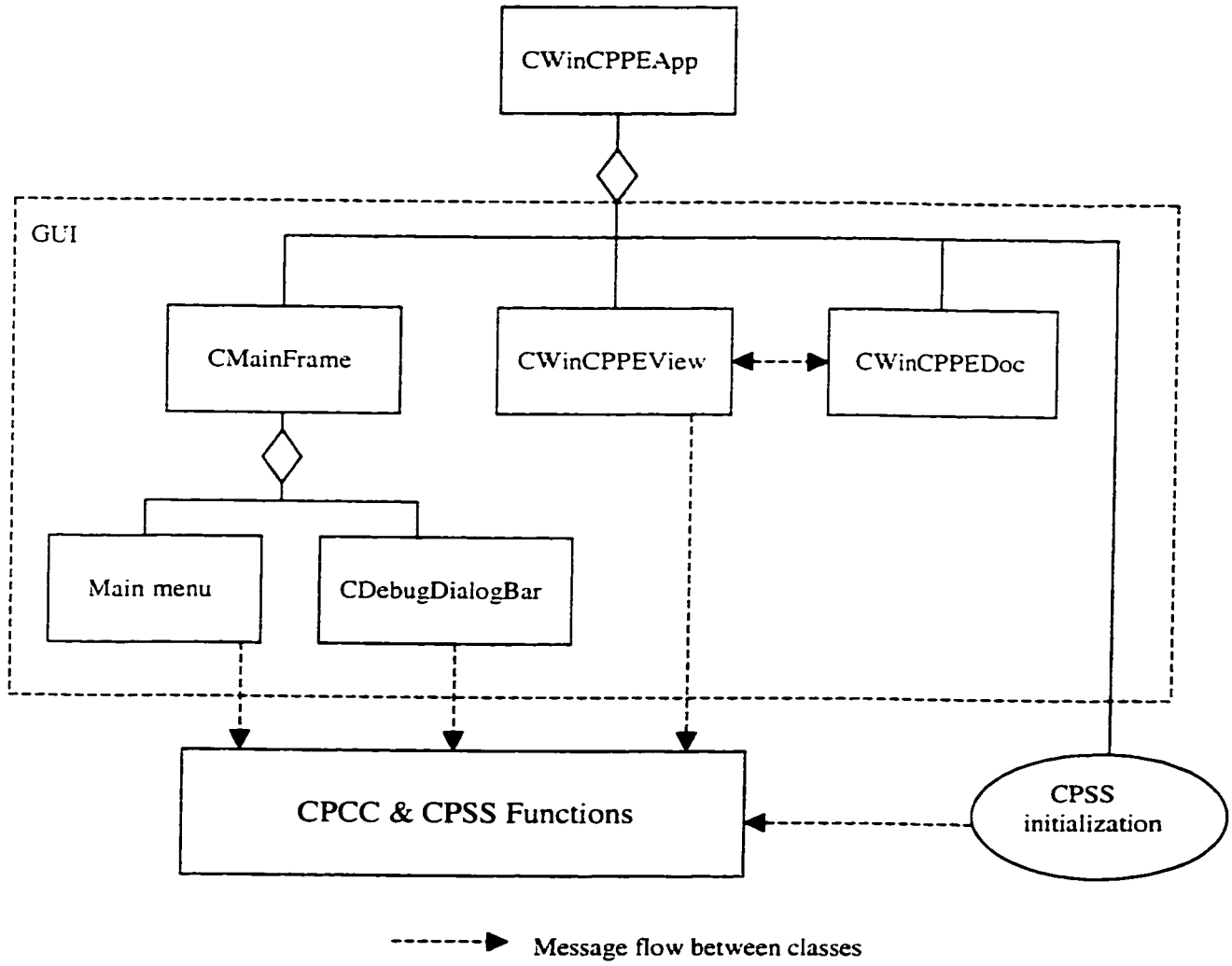


Figure 27 : Major classes and their relationship in CPPE GUI

CWinCPPEApp (file winCPPE.h, winCPPE.cpp)

This is the application object based on the CWinApp class in MFC. It is the heart of an MFC application program, since CWinApp provides member functions for initializing the application program. CWinApp also provides function that can be overridden to customize the application program's behavior. CWinCPPEApp overrides the function InitInstance() inherited from CWinApp to initialize the CPPE application, including the creation of the application window, and the initialization for CPSS. InitInstance() creates the application window by instantiating classes CMainFrame, CWinCPPEDoc and CWinCPPEView, and the functionality of these classes are introduced in the following paragraphs.

The application object of CWinCPPEApp is declared with global scope so that it will be instantiated in memory at the very outset of the program.

CMainFrame (file MainFrm.h, MainFrm.cpp)

This application class is inherited from the MFC's class CFrameWnd. CFrameWnd provides functions that models basic features of a window frame. CMainFrame overrides the function OnCreate() inherited from CFrameWnd to customize the window's "look" and behavior for CPPE.

The window provides mainly two types of interface components for user interaction: a menu bar and a debugging dialog bar. The behavior of the window is implemented by message mapping, MFC's way of relating user's input from the interface to the application's internal functionality. Each menu item or button in the debugging

dialog bar is associated with a message type, which will invoke a CPPE function registered with this type of message. Menu bar is implemented in an application program's resource file winCPPE.rc. The debugging dialog bar is implemented by another class CDebugDialogBar.

CDebugDialogBar (file DebugDialogBar.h, DebugDialogBar.cpp)

This class is derived from the MFC's class CDialogBar. The CDialogBar class provides the functionality of a Windows modeless dialog box in a control bar. A dialog bar resembles a dialog box in that it contains standard Windows controls. Our debug dialog bar is used to provide function buttons for the debugging tools in CPPE. Each button is related to a message type which will be mapped to a debugging function in CPPE. User can use most of the debugging tools from the debug dialog bar.

CWinCPPEDoc (winCPPEDoc.h, winCPPEDoc.cpp)

This class is derived from the MFC's class CDocumnet. MFC's document class provides a template to store the application's data, a visible representation of what appears in the GUI. It also provides public member functions that other objects can use to query and modify its data.

CWinCPPEView (winCPPEView.h, winCPPEView.cpp)

This class is derived from the MFC's class CEditView. MFC's view class offers two major functions: to render visual representation of a document's data on the screen, and

to translate the user's input into messages that will operate on the document's data or invoke other application functionality such as the functions of CPCC or CPSS.

Chapter 6

Example Applications of Debugging Tools

The effectiveness of the debugging tools is demonstrated by actually running some parallel application programs and analyzing the debugging output and performance statistics. The performance debugging tools allow us to effectively simulate different parallel topologies, communication parameters, and virtual-to-physical-architecture mappings. Based on the study of the performance affecting factors in parallel computing reviewed in Chapter 2, we can evaluate the effectiveness of our debugging tools, remedy the defects and recommend enhancement for our future work.

The matrix multiplication program (Figure 28) is presented here to illustrate the application of performance debugging tools in the process of parallel program development. In this program, the user specifies the virtual architecture as a 2D torus and the physical architecture as a 2D mesh. However, the simulator can override the physical architecture using the debugging tool we developed before starting execution, without recompiling the application program. The program first initializes two 2D arrays. In order to eliminate the effect of the sequential initialization cost on the overall performance, we use a function timeOff() to turn off the global clock for the phase of array initialization. We then turn on the global clock and start the phase of parallel computing. In this way, the performance statistics can truly reflect the performance of parallel execution. The **forall** statement in function pMultify() has three parts. The first part indicates the lower

and upper bounds of the indices. The second part indicates the absolute virtual processor ID on which the process is spawned, and its associated channel variables. The last part is the function to be invoked.

```
#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 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]);
}

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
}
```

Figure 28 : Matrix multiplication on an 8x8 torus

The following sections demonstrate the application of performance debugging tools in the study of the relationship between performance and performance affecting factors.

6.1 Virtual-to-physical Mapping And Performance

Virtual Topology	Physical Topology	Mapping	SeqExe Time	ParExe Time	Speedup
8x8 Torus	8x8 Mesh	Default	35816	3630	9.87
8x8 Torus	8x8 Mesh	Identity	35816	3630	9.87
8x8 Torus	8x8 Mesh	Random	35886	4910	7.31
8x8 Torus	8x8 Mesh	Torus-to-Mesh	35771	3490	10.25
16x16 Torus	16x16 Mesh	Default	265766	14580	18.23
16x16 Torus	16x16 Mesh	Identity	265766	14580	18.23
16x16 Torus	16x16 Mesh	Random	266471	23150	11.51
16x16 Torus	16x16 Mesh	Torus-to-Mesh	265971	14960	17.78

Table 4: Performance statistics for virtual-to-physical-architecture mapping

Table 4 is the performance statistics for different types of virtual-to-physical mapping, using wormhole routing.

- According to our design, when the number of virtual processors is equal to the number of physical processors, the default mapping is effectively the same as the identity mapping. This is verified by the execution results.

- For a wormhole-routed parallel computer, although the network latency is no longer affected by the distance, the link contention can still affect the communication efficiency [24][26]. From the execution result, when the size of the network becomes larger, the parallel speedup for the random mapping is significantly degraded compared with the identity, default or Torus-to-Mesh mapping, where the link contention is optimized because of the nature of the mapping.

6.2 Topology And Performance

Virtual Topology	Physical Topology	Mapping	SeqExe Time	ParExe Time	Speedup
4x4 Tourse	4x4 Torus	Default	5127	700	7.32
4x4 Tourse	4x4 Mesh	Default	5135	1010	5.08
4x4 Tourse	Ring [256]	Default	5146	3820	1.35

Table 5: Performance statistics for topologies

Table 5 is the performance statistics for different parallel topologies, using wormhole routing. If the physical topology is the same as the virtual topology, which is defined in the application program as the optimized topology for the application, it will gain the maximum benefit from parallel computing, showing the best parallel speedup. With the physical topologies increasingly differentiate from the virtual topology, the parallel speedup is degraded.

6.3 Communication Parameters And Performance

Virtual Topology	Physical Topology	Packet Size(bytes)	SeqExe Time	ParExe Time	Speedup
16x16 Tourse	16x16 Mesh	4	265766	14580	18.23
16x16 Tourse	16x16 Mesh	8	266680	27560	9.68
16x16 Tourse	16x16 Mesh	16	266845	44020	6.06

Table 6: Performance statistics for packet size

Table 6 is the performance statistics for different packet sizes, using wormhole routing. For wormhole routing, the network latency is affected by the packet size. It is verified by the execution results, with the parallel speedup degraded when the packet size increases.

Chapter 7

Conclusion and Future Work

In this thesis, we presented the design and implementation of the Visual Performance Debugger for CPPE. The Visual Performance Debugger aims to provide flexible and efficient software tools for developing parallel applications and optimizing their performance. It will be useful for the parallel application programmers to design, implement, and test their programs to improve performance before working with the actual parallel systems. Through practice with coding and testing parallel programs they will develop a practical skill of isolating and removing performance bottlenecks that may severely limit the parallelism achieved by the programs. Within the research community, the visual performance debugger will be a helpful tool for the testing and analysis of new algorithms in parallel computing and virtual-to-physical architecture mapping.

Bearing this objective in mind, we have conducted a comprehensive survey on the practical sources of performance degradation both in inter-processor communication and parallel computing. Existing simulation systems have also been studied to find out pros and cons of their debugging features.

Our visual performance debugger makes the CPPE an excellent environment for developing and fine-turning parallel programs due to the following advantageous features:

- Flexibility: The debugger provides a rich set of correctness and performance debugging tools. In addition to conventional sequential debugging functions, user can set breakpoints and trace variables based on individual parallel process during execution. The performance debugging tools allow user to simulate a wide range of parallel topologies, sizes, communication parameters, and virtual-to-physical-architecture mapping.

- Informative: The performance debugging tools provide debugging information and performance statistics at various levels of details. These information well address the relationship between the program performance and the corresponding performance affecting factors, which significantly help the user to understand the behavior of the parallel programs in order to fine-turning their programs.

- Accurate: Simulation results are based on the functional simulation technique which can provide the most accurate performance data among existing simulation techniques. In addition, performance debugging tools enable the user to reconfigure network parameters to accurately simulate a particular parallel computer system.

- Repeatable: Simulation results obtained from the debugging tools are repeatable. Therefore it provides a stable and reliable debugging environment. At the same time, multiple execution of a non-deterministic application is also supported.

- Portable and user friendly: The graphic user interface enables the user to easily use the debugging tools and analyze the debugging information and performance

statistics. Compile flags are used to make the visual performance debugger portable to different platforms.

The visual performance debugger is designed in a modular fashion so that it will be easy to extend its functionality. The following debugging features can be implemented in the future to further enhance the functionality of the visual performance debugger:

- Capturing and rerunning a debugging session: sometimes the user may want to rerun a debugging session to better understand some critical behaviors of the program. User should be allowed to go back to a certain point and replay the subsequent session.

- Saving and restoring the debugger state: a similar functionality would allow the user to save the current set of debugging state such as breakpoints, traced variables into a file. Later on, the user can restore this information when re-invoking the debugger and replay the previous debugging session.

- Tracing function or procedure calls: when a program contains several functions or procedure calls, the user may want to know the sequence of calls that led to the current point of suspension.

- With the evolution of technology in the design of graphical user interface, we can continuously employ new features to enhance the graphical functionality of the debugger such as 3D display of 2D and 3D topologies, visualization of message flow, and visualization of stepping through a program execution.

Appendix A

CPPE User's Manual For UNIX Platform

1 Configuration

This section introduces the procedure to set up the environment and start the program. CPPE is currently compiled on SunOS 5.5 SPARC machine. The executable program name is cppe.

1.1 Environment Files

CPPE has two environment files: resource file CPPE and config file cppe_motif.cfg. Normally, user should install these two environment files in the same working directory as CPPE executable. But these two environment files are both optional. If they are not provided, CPPE will use default resource and configuration to start.

Resource File CPPE

CPPE needs a resource file called CPPE. If C shell is the default shell on your machine, You need to add the following line to your .cshrc file to specify the path of the resource file:

```
setenv XAPPLRESDIR $WORKING_DIR
```

```
e.g. setenv XAPPLRESDIR /home/grad/John/cppe
```

where WORKING_DIR is the path where the resource file CPPE exists. For different shell, you need to set the environment properly following the corresponding shell syntax.

Configuration File cppe_motif.cfg

CPPE has an optional configuration file called cppe_motif.cfg in the same directory as the CPPE executable. When CPPE startes, it looks for this configuration file to initialize the working environment of CPPE, including the current working directory, the editor type, the current physical architecture of the simulated parallel system and the network type. User can change the initial CPPE configuration by modifying file cppe_motif.cfg. When modifying cppe_motif.cfg, make sure to follow the existing format in the initial cppe_motif.cfg file.

If the configuration file cppe_motif.cfg is not found in the directory where CPPE starts, CPPE starts with a default configuration. Users then have the option to save a current working configuration as a default configuration and a default configuration file cppe_motif.cfg will be created in the directory where CPPE starts.

1.2 Environment Variables

CPPE needs another environment variable named "CPPE". It is used to define the path to find the input .c and .h files in the compile and execution process. You need to add the following line to your .cshrc file to set this environment variable:

```
setenv CPPE $SOURCE_DIR
```

```
e.g. setenv CPPE /user/John/cppe/src:/user/John/cppe
```

where SOURCE_DIR is the path where the source .c and .h files are stored. For different shell, you need to set the environment properly following the corresponding shell syntax

1.3 Start The CPPE Program

To start the CPPE program, at the command line, type

> cppe

1.4 Configuration Setup

When CPPE is launched, a default configuration is set up based on the environment variables and the default configuration file. In the process of program development, users can reconfigure the CPPE execution environment based on their needs for debugging and performance tuning. CPPE provides a functionality that users can save the current configuration to a data file and this configuration is retrievable in the future, so that CPPE can be easily configured to meet the specific requirement of different users.

CPPE allows users to save the current configuration as a default configuration or as a specific debugging configuration. A default configuration can be used to initialize the working environment when a new CPPE session is started or reset the working environment to its default state during an execution session. The data that are saved in a default configuration include the current working directory, the editor type, the current physical topology, all the available physical topologies including those that are defined by a user at run time, and the current network type. A specific debugging configuration

will also include the user-defined physical-to-virtual topology mapping function, if defined, in addition to the other configuration data found in a default configuration file.

To save the current configuration as default configuration, from the main menu click the menu pane **Configure**, then select the menu item **Save Configure As Default**. To save the current configuration as a specific debugging configuration, from the main menu click the menu pane **Configure**, then select the menu item **Save Configure As Other**. A file save dialog box will be popped up for users to specify a file name. To reset the default configuration, from the main menu click the menu pane **Configure**, then select the menu item **Load Configure From Default**. To reset a specific debugging configuration, from the main menu click the menu pane **Configure**, then select the menu item **Load Configure From Other**. A file selection dialog box will be popped up for users to provide the configuration file name.

2 CPPE Functionality

This section introduces the utilities supported by CPPE and the usage during execution. CPPE contains three major functionalities: parallel application program compiling, parallel application program execution, correctness and performance debugging.

A graphic user interface (GUI) is popped up when the user starts the CPPE program as described in section 1.3. The GUI main frame is composed of main menu, option menus, function buttons, output window and message line. All CPPE functions can be invoked either from main menu, or option menus, or function buttons.

2.1 Create Application Program Source File

A parallel application program source file should be created first as a text file, with file name extension ".c". The user can use any text editor to create the source file outside the CPPE program. CPPE also provides a utility to invoke a text editor from inside the CPPE.

To invoke the text editor from inside CPPE, from the main menu in the GUI main frame, click the **File** menu pane, select menu item **Edit**. Depending on the configuration, either a vi editor or a emacs editor will be invoked. The editor starts with the source file currently opened in CPPE. If no source file has yet been opened, a file named "untitled" will be opened in the editor. The text editor runs in background so that it can work in parallel with the CPPE program.

2.2 Compile Application Program Source File

Open A Source File

To open a source file, click the function button **Open** in the GUI main frame. A file selection dialog box is popped up. To select a directory, double click the requested directory in the **Deirectoies** list. Then the files in that directory will be displayed in the **Files** list besides the **Directories** list. To specify a filter, modify the filter in the **Filter** field. To select a file, single click the requested file, then click **OK**. The file name will be displayed in the **Source** option menu, and the working path will be displayed in the **message line** at the bottom of the main frame.

Compile a Source File

Compile process is used to compile an application program source file into a virtual machine code (vCode) file, with file name extension ".cod", for execution in CPPE.

A source file should be opened before it can be compiled. To compile a source file, click the function button **Compile**. If compile succeeded, the corresponding code file name is displayed in the **Cod** option menu. At the same time, the virtual architecture of the program will be displayed in the **Virtual Arch** text field and the default mapping will be displayed in the **Mapping** option menu.

Open a vCode File

Function button **Open** can also be used to open a vCode file directly. The opened vCode file name is displayed in the **Cod** option menu. At the same time, the virtual architecture of the program will be displayed in the **Virtual Arch** text field and the default mapping will be displayed in the **Mapping** option menu.The opened code file is immediately ready for execution.

Close a source file or vCode file

The number of source files or vCode files that can be opened at the same time is limited in CPPE. You may want to close some currently opened files in order to open other files. To close a source file or vCode file, click the function button **Close** in the main frame. A dialog box will pop up for user to select the type of file to be closed. The user has the option to close the currently selected source file in the **Source File** option menu in the main frame, the currently selected vCode file in the **Executable File** option menu in the main frame, or both at the same time.

2.3 Execution And Debugging

2.3.1 Execution

After a source file is successfully compiled or a vCode file is opened, click the function button **Run** in the main frame. The execution result and any debugging message will be displayed in the main window of the main frame.

2.3.2 Viewing Source Code and vCode

After a source file is successfully compiled or a vCode file is opened, or after program execution is suspended, user can specify a range of source code or vCode to be displayed by referring to the line numbers.

To display the source code, from the main menu in the main frame, click the **Source** pane and select the **List Source Code** menu item. A new **List** window will be popped up. In the **List** window, specify the line numbers in the **List From** and **To** fields and then click the **List** button. If no line numbers are specified in the **List From** and **To** fields, the whole source file will be displayed. To close the **List** window, click the **Back** button in the **List** window.

To display the vCode, from the main menu in the main frame, click the **Source** pane and select the **List vCode** menu item. A new **vCode** window will be popped up. In the **vCode** window, specify in the **List From** and **To** fields the starting and ending line numbers for the vCode to be displayed, and then click the **List** button. If no line numbers are specified in the **List From** and **To** fields, the whole vCode file will be displayed.

2.3.3 Setting Breakpoints

The user can set breakpoints on executable instructions in the source program to automatically interrupt the program execution. This function is useful for helping the user to locate bugs in the program.

Breakpoints are set in the CPPE by referring to program line numbers. To set a breakpoint, click the function button **Break** in the main frame. A new window titled **List** will be popped up. From the **List** window, the user gets a list of program source code in a source code list. To set a breakpoint, single click on the source line in the source list where the break point will be. Then click the **Break** button in the **List** window. The selected break point line will be displayed in the breakpoint list below the source code list. To clear a breakpoint, single click on the breakpoint line in the break point list, then click **Unbreak** button in the **List** window. Both setting a breakpoint and clearing a breakpoint can also be done by double clicking the source line in the source list or in the breakpoint list.

If a breakpoint is set and CPPE is set to Debug mode, when users execute an application program, the execution will stop at the breakpoint and a new window titled **Step Execution From Breakpoint** will pop up showing the program source code with the breakpoint highlighted.

2.3.4 Stepping Through A Process

When any running process tries to execute a line in a program with a breakpoint, the whole program execution will be suspended. At this point, the execution of the program may be continued with two functions: Continue or Step. If Continue function is used, the

execution will be continued until the next breakpoint is encountered by any process. If Step function is used, the execution will be continued line by line from the breakpoint in the suspended process. User can also specify the number of lines in each step and set the "Step-Process" to a different running process. CPPE provides these two functions so that the user can step through the execution of a program in order to trace the execution status and debug the program.

The Step function counts just executable lines in the program listing, independent of how many statements are contained on a given line. For program loops, the total number of lines executed is counted until the number reaches the number specified by the Step function. Although the Step function is applied to the currently suspended process, all the other parallel processes also continue to execute in parallel, so they also will be advancing in execution.

To use Continue function, click the function button **Continue** in the main frame. To use the Step function, if user wants to step through the currently suspended process line by line, click the function button **Step** in the main frame. In either case, if the program execution stops at a new breakpoint of execution, the new breakpoint of source code will be highlighted in the **Step Execution From Breakpoint** window. When program execution terminates, the **Step Execution From Breakpoint** window will be shut down.

To specify a different "Step-Process", from the main menu, click the **Debug** menu pane, select the **Step** menu item, then select the **Set Step Process** submenu item. To step a process with a different number of lines in each step, select the same **Step** menu item, then select the **Step** submenu item.

2.3.5 Tracing Variables

Whenever the execution of the parallel program is suspended, the user may want to examine the current value of variables in the current environment of each process. CPPE provides two functions for this purpose: Show and Trace.

Function Show is used to display the value of a variable when program execution is in suspension state. User can use this function to examine variables in the current environment of each active process. Active processes are those that may be in state of Ready, Running, Blocked, Delayed or Spinning.

To use function Show, the variable should be in a currently active process. To get the list of active processes, from the main menu, click menu pane **Report**, select menu item **Status**, the default function of Status will give a full list of process status. To use Show function, click the function button **Show** in the main frame window, specify a variable name and an active process id. If the variable is an array, user should specify the index range that user wants to display. Then click the button **OK**. The output will be displayed in the output window in the main frame.

Function Trace is used to trace a particular variable during the execution process. User essentially sets a flag on the traced variable. Whenever that variable is referenced during subsequent program execution, the program will be suspended as it is for breakpoints. The traced variable may be referenced by many different processes in a variety of locations in the program.

To use function Trace, from the main menu, click menu pane **Debug**, select menu item **Trace**. A trace dialog box will be popped up. Specify the variable name and process

id in the trace dialog box and then click button **Trace**. The traced variables are displayed in the **Trace Variable List** in the dialog box. User can clear a trace variable later by selecting the variable from the **Traced Variable List** and clicking the button **UnTrace**.

Users have the option to turn on or off the Trace function before or during program execution. There is a group of **Trace On/Off** radio buttons in the main frame window for user to turn Trace function on or off.

2.3.6 Alarm

Alarm is used to suspend the program execution when a certain amount of time is reached. The functionality of Alarm is similar to setting a breakpoint so that user can examine execution status in the process of program execution.

To set an alarm, from the main menu in the main frame, click the menu pane **Debug** and select the menu item **Alarm**. An **alarm_popup** dialog box will pop up. User can turn the alarm function on or off from this dialog box. When the alarm is turned on, user can specify the alarm time in the text field Enter Alarm Time.

2.4 Network Architectures and Mapping

2.4.1 Specifying the Architecture

When CPPE starts, the default architecture is a 2D-mesh parallel computer with size of each dimension being 4 (mesh 4x4). The user may override this default and specify a wide range of other architectures, including many of the common parallel topologies. This allows the performance of the parallel program to be simulated and evaluated on a wide range of parallel computer architectures according to the choice of the users.

CPPE has predefined some most common architectures in the system, which can be used directly by selecting from the option menu **PhyArch** in the main frame. User can also define new architectures according to the needs of their applications. To define a new architecture, from the main menu, click the menu pane **Network**, then select the menu item **Architecture**. An **archDialog** dialog box will pop up. A new architecture is defined by architecture type and size. Select an architecture type from the option menu **phyArch** in the dialog box. Then specify the architecture size in the text field in the dialog box. For architecture of type Line, Ring, Fullconnect and Shared, specify the architecture size by entering the number of physical processors. For a multi-dimensional architecture, such as mesh, torus, enter the size of each dimension separated by commas. For a hypercube architecture, enter the number of dimensions. The newly defined architecture will be added to the **phyArch** option menu in the main frame and becomes the current parallel system architecture.

2.4.2 Virtual-to-Physical Architecture Mapping

Message-passing parallel programs are encouraged to be written using virtual topologies, the topologies most natural to express the program communication structure. However, the virtual topology may be the same as or different from the topology of the physical system on which the program is running. CPPE supports virtual-to-physical architecture mapping. The objectives of virtual-to-physical architecture mapping are to minimize communication cost by minimizing the distance between communicating processes, and to balance the workload among physical processors.

CPPE currently supports six types of mapping: Default, Identity, Random, Ring-to-Line, Torus-to-Mesh and User-defined Mapping. In Random mapping, virtual processors are randomly assigned to physical processors at run-time. In Identity mapping, the virtual topology are mapped to the identical physical topology, which is the exact match between virtual topology used in programming and the actual underlying physical machine. Both Random and Identity mapping can only happen when the number of virtual processors is no greater than the number of physical processors. When the number of virtual processors is greater than the number of physical processors, effectively these two mappings will fall into Default mapping, where virtual processors are divided into blocks, and each block is then mapped to the physical processors. Ring-to-Line mapping is available when the virtual architecture is Ring, the physical architecture is Ling and the number of virtual processors equals to the number of physical processors. Torus-to-Mesh mapping is available when virtual architecture is Torus, the physical architecture is Mesh and these two architectures are identical except the wraparound links. A user-defined mapping allows unlimited mapping functions to be specified at debugging time to satisfy the need of different user applications.

To specify a virtual-to-physical mapping, select a mapping type from the option menu **Mapping** in the main frame.

To specify a user defined mapping function, users should enter a mapping function string following the syntax defined in CPPE. In general, a mapping function string has the following format:

$$\texttt{mapfunction}(x_1, x_2, \ldots, x_m) = [y_1, y_2, \ldots, y_n].$$

which maps virtual processor (x_1, x_2, ..., x_m) to physical processor (y_1, y_2, ..., y_n). Most C expressions can be used inside the square brackets.

As an example, a one-to-one mapping from a two-dimensional torus to a two-dimensional mesh can be translated as following:

```
torus2mesh(x, y) = [x<=(#1-1)/2 ? 2*x : 2*(#1-x)-1,
                    y<=(#2-1)/2 ? 2*y : 2*(#2-y)-1].
```

where *#n* denotes the size of the n^{th} dimension of the virtual architecture, and the pair (*x*, *y*) denotes the Cartesian coordinate of the virtual processor in the virtual architecture. The square brackets enclose the right-hand side of the mapping function. The pair of values inside the square brackets denotes the Cartesian coordinate of the physical processor in the physical architecture. The final dot is used to signal the end of the mapping string so that users can break the map string in several lines during inputting.

A many–to-one mapping from a two-dimensional torus to a two-dimensional mesh can be expressed as follwoing:

```
torus2mesh(x,y)=
    [x<=(#1-1)/2 ? (2*x)/(#1/$1):(2*(#1-x)-1)/(#1/$1),
    y<=(#2-1)/2 ? (2*y)/(#2/$2):(2*(#2-y)-1)/(#2/$2)].
```

where *$n* denotes the size of the n^{th} dimension in the physical architecture. The denominator (*#i/$i*) in the physical Cartesian coordinate provides the effect of block cyclic mapping.

To specify the user-defined mapping function, from the **Mapping** option menu in the main frame, select the **User Defined** menu item. A **User Defined Mapping** dialog

box will pop up. The user can either enter a mapping function string in the text field in the dialog box, or load a mapping function string from a file that contains a mapping function string. To load a mapping function string from a file, click the button **Load** in the dialog box and a file selection box will pop up. Select the file that contains the mapping function string and click button **OK** in the file selection box. The mapping function string will be loaded into the text field in the **User Defined Mapping** dialog box. The user can also save the user input mapping function string into a file. The button **Save** in the **User Defined Mapping** dialog box is used for this purpose.

2.4.3 Network Routing Type

CPPE can simulate different network types. Currently it supports packet switching network, simulated packet switching network, shortest path network and wormhole-routed network. In packet switching network, basic communication delay is adequate to reflect the communication delay when the message traffic is low enough that there is no interference between messages that might result in congestion delays. Some programs have more frequent communication that travel longer paths in the network, resulting in the potential of message congestion and further communication delay. In order to simulate the execution of different parallel programs, CPPE provides an option to turn on or off the message congestion. The simulated packet switching network simulates a packet switching network with message congestion turned on. The shortest path network simulates a packet switching network with message congestion turned off.

Users can select a network type before program execution starts or change network type during program execution. The network type can be selected from the option menu **Network** in the main frame.

2.4.4 Communication Delay

CPPE can simulate the parallel architecture, using message passing for inter-process communication. An important parameter for the inter-process communication is the Basic Communication Delay, which is the basic time to communicate a message packet between two processors with a direct physical communication link. User can define the value of the Basic Communication Delay at run time. From the main menu, click the menu pane **Network**, select the menu item **CommDelay**. A **comm_delay_popup** dialog box will pop up for user to enter the value of Basic Communication Delay, which is defined as the number of time units.

2.4.5 Communication Parameters

In a wormhole-routed network, users can check the current setting of communication parameters and redefine the values of communication parameters before simulation starts or when program execution is in suspension. The following communication parameters are defined in CPPE which are configurable during simulation: number of lane per channel, flit size (bytes), packet size (bytes), buffer size (flits), startup overhead per message (time units), startup overhead per packet (time units), headOtherFlitSpeedRatio (speed ratio between head flit and following flits). To display the current setting of communication parameters for the current routing option, from the main menu, click the

menu pane **Architecture**, select the menu item **Network Parameters**, then select the cascaded menu item **Display**. The current values of the communication parameters will be displayed in the output window in the main frame. To redefine the communication parameters, from the main menu, click the menu pane **Architecture**, select the menu item **Network Parameters**, then select the cascaded menu item **Change**, a **Change Network Parameters** dialog box will pop up. The current values of all the available communication parameters are displayed in the dialog box. To change the value of a parameter, click the corresponding text field for that parameter and reenter the value, then click button **OK** in the dialog box.

2.4.6 Vary Processor Speed

This function is used for testing multiple executions of non-deterministic applications and robustness of deterministic programs, as discussed in section 4.2.3. Race conditions are simulated by varying relative processor speeds. When the Vary Processor Speed option is turned on, users need to provide an integer Random Number Seed, which will be used to create a random number r_i between 0 and 1 (>0) for each physical processor i that will be used to increase the speed by a factor of $1/r_i$. This randomly selected speed factor for each processor will remain in effect throughout the subsequent program execution. The particular random speed factors chosen completely dependent on the Random Number Seed: using the same seed again will result in the same set of processor speed factors.

To turn on the Vary Processor Speed option, from the main menu, click the menu pane **Architecture**, then select the menu item **Vary Processor Speed**, and a **Set Vary Processor Speed** dialog box will pop up. Set the **Vary Speed** on or off from the radio

selection button. When the option is set on, the user can specify an integer number of Random Number Seed in the text field in the dialog box, and click button **OK**.

2.5 Program Performance Statistics

When CPPE executes a program, it keeps track of the relative timing of all processes and generates a range of performance statistics at the end of execution to help the user understand the behavior and evaluate the performance of the program.

2.5.1 Execution Time

CPPE is a code interpreter. The program source code is first compiled into a virtual-machine code (vCode) which is interpreted rather than directly executed. When CPPE runs a program, it interprets the vCode and uses an estimated execution time for each vCode instruction and keep a total execution time of the program. The estimated execution time differs between instructions depending on the complexity of the instructions. Using this estimated execution time, CPPE can simulate the performance of the program on a real multiprocessor or multicomputer.

At the end of execution, CPPE will display the total Sequential Execution Time and the total Parallel Execution Time. Sequential Execution Time is the estimated execution time on a uniprocessor computer. Parallel Execution Time is the estimated execution time on an actual target multicomputer or multiprocessor. From the ratio of sequential/parallel execution time, user can estimate the performance improvement by parallel computing.

2.5.2 Time

Time function can be used whenever program execution is suspended to give the total elapsed time since the beginning of the program execution.

To use the time function, from the main menu in the main frame, click the menu pane **Report** menu, select the menu item **Time**. The output will be displayed in the main output window in the main frame.

2.5.3 Utilization

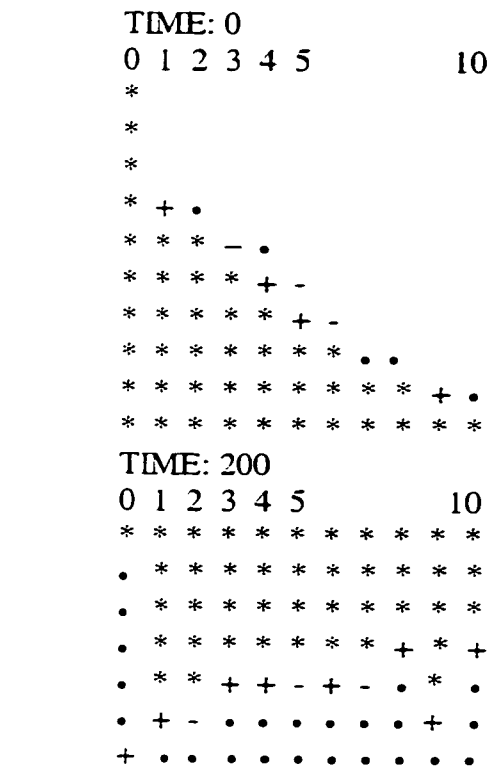
Utilization function is used to show the usage of physical processors in a particular parallel architecture. The utilization of a given physical processor is defined as the proportion of the time the processor is actually running. Whenever the program execution is suspended, the user can use this function to display the utilization of a range of processors.

To use the Utilization function, from the main menu in the main frame, click the menu pane **Report**, select the menu item **Utilization**. A **utilizationDialog** dialog box will pop up. The user can specify the range of processors that the user wish to see the utilization value from the dialog box.

2.5.4 Program Performance Profile

CPPE can create a visual performance profile that will help the user to understand the program performance. The visual performance profile will be displayed in the main output window in the main frame. It shows the processor utilization during successive time intervals of program execution.

A sample profile fragment is shown as following:



In the sample profile fragment, the first line shows the current elapsed program execution time. The second line shows the processor number. Time advances vertically down the page from top to bottom, with each successive line represents a time interval of duration 10 time units. The marks indicate the processor utilization during each time interval: "*" indicates 75-100 percent utilization, "+" indicates 50-75 percent utilization, "-" indicates 25-50 percent utilization, and "." indicates 0-25 percent utilization.

To create a performance profile, the user needs to turn on the profile option. In the CPPE main frame there is a group of **Profile** radio buttons that let users to turn on or off the profile option. The default range of processors in the profile is all processors used in

the program. The default time interval in the profile is 10 time units. User can also specify a different range of processors and time interval. From the main menu of the main frame, click the menu pane **Report**, select the menu item **Profile**, a **profileDialog** dialog box will pop up. User can specify the range of processors and the time interval from this dialog box.

Appendix B

CPPE User's Manual For Windows Platform

1 Configuration

1.1 Configuration File

CPPE has an optional configuration file called cppe_win.cfg in the same directory as the CPPE executable. When CPPE starts, it looks for this configuration file to initialize the working environment of CPPE, including the current working directory, the editor type, the current physical architecture of the simulated parallel system and the network type. User can change the initial CPPE configuration by modifying file cppe_win.cfg. When modifying cppe_win.cfg, make sure to follow the existing format in the initial cppe_win.cfg file.

If the configuration file cppe_win.cfg is not found in the directory where CPPE starts, CPPE starts with a default configuration. Users then have the option to save the current working configuration as a default configuration and a default configuration file cppe_win.cfg will be created in the directory where CPPE starts.

1.2 Environment variables

We can also define environment variables in the system's autoexec batch file. The following environment variables are defined in file autoexec.bat:

```
set CPPE=.;D:\cppe98\test
set PATH=.;C:\tools;D:\cppe98\winCPPE\Debug
```

CPPE is used to define the path to find the input .c and .h files in the compile and execution process. PATH is used to define the path to launch the CPPE executable from DOS command line.

1.3 Configuration Setup

When CPPE is launched, a default configuration is set up based on the environment variables and the default configuration file. In the process of program development, users can reconfigure the CPPE execution environment based on their needs for debugging and performance tuning. CPPE provides a functionality that users can save the current configuration to a data file and this configuration is retrievable in the future, so that the CPPE can be easily configured to meet the specific requirement of different users.

CPPE allows users to save the current configuration as a default configuration or as a specific debugging configuration. A default configuration can be used to initialize the working environment when a new CPPE session is started or reset the working environment to its default state during a CPPE session. The data that are saved in a default configuration include the current working directory, the editor type, the current physical topology, all the available physical topologies including those that are defined by a user at run time and the current network type. A specific debugging configuration

will also include the user-defined physical-to-virtual topology mapping function, if defined, in addition to the other configuration data found in a default configuration file.

To save the current configuration as default configuration, from the main menu click the menu pane **Configure**, then select the menu item **Save Configure As Default**. To save the current configuration as a specific debugging configuration, from the main menu click the menu pane **Configure**, then select the menu item **Save Configure As Other**. A file save dialog box will be popped up for users to specify a file name. To reset the default configuration, from the main menu click the menu pane **Configure**, then select the menu item **Load Configure From Default**. To reset a specific debugging configuration, from the main menu click the menu pane **Configure**, then select the menu item **Load Configure From Other**. A file selection dialog box will pop up for users to provide the configuration file name.

2 CPPE Functionality

This section introduces the utilities supported by CPPE and the usage during execution. CPPE contains three major functionalities: parallel application program compiling, parallel application program execution, correctness and performance debugging.

A graphic user interface (GUI) will pop up when the user starts the CPPE program from Windows platform. The GUI main frame is composed of main menu, option menus, function icons and output window. All CPPE functions can be invoked either from main menu, or option menus, or function buttons. When click on the function icon **D** in the main frame, a debugging menu bar will be popped up which contains function buttons for all the debugging utilities.

2.1 Create Application Program Source File

A parallel application program source file should be created first as a text file with file name having the extension ".c" . Users can use any text editor provided in the PC platform to create the source file outside the CPPE program. Or users can invoke a default text file editor from the **Edit** menu in the CPPE main frame.

2.2 Compile Application Program Source File

Open a Source File

To open a source file, click the standard Windows' file **Open** icon in the main frame. A standard Windows' file selection dialog box will pop up. Select the application source file and click **Open** button in the dialog box. The file name will be displayed in the **Source File** option menu.

Compile a Source File

Compile is used to compile an application program source file into virtual machine code (vCode) file for execution in CPPE.

A source file should be opened before it can be compiled. To compile a source file, click the function button **C** in the main frame. If compile succeeded, the corresponding vCode file name is displayed in the **Execution File** option menu. At the same time, the virtual architecture of the program will be displayed in the **Virtual Arch** text field and the default mapping will be displayed in the **Mapping** option menu. Compile function can also be invoked from the menu item **CPCC** in the main menu **File**.

Open a vCode File

The **Open** icon can also be used to open a vCode file directly. The opened vCode file name is displayed in the **Execution File** option menu. At the same time, the virtual architecture of the program will be displayed in the **Virtual Arch** text field and the default mapping will be displayed in the **Mapping** option menu. The opened vCode file is immediately ready for execution.

Close a source file or vCode file

The number of source files or vCode files that can be opened at the same time is limited in CPPE. You may want to close some currently opened files in order to open other files. To close a source file, from the main menu click the menu pane **File**, select the menu item **Close Source File**. The currently selected source file in the **Source File** option menu in the main frame will be closed. To close a vCode file, from the main menu click the menu pane **File**, select the menu item **Close vCode File**. The currently selected vCode file in the **Executable File** option menu in the main frame will be closed.

2.3 Execution And Debugging

2.3.1 Execution

After a source file is successfully compiled or a vCode file is opened, click the function icon **E** in the main frame. The execution result and any debugging message will be displayed in the output window of the main frame.

2.3.2 Open the Debugging Menu Bar

The **Debugging Menu Bar** contains all the debugging function buttons in CPPE. User can also turn on or off the debugging mode for program execution from this menu bar. Open the Debugging Menu Bar by clicking the function icon **D** in the main frame.

2.3.3 Viewing Source Code And vCode

After a source file is successfully compiled or a vCode file is opened, or after program execution is suspended, user can specify a range of source code or vCode to be displayed by referring to the line numbers.

To display the source code, from the Debugging Menu Bar, click the **SrcCode** button, a dialog box will pop up. In the dialog box, specify the starting and ending line number of the source code to be displayed. Then click the **OK** button in the dialog box. The source code will be displayed in a separate **Source Code Window**.

To display the vCode, from the Debugging Menu Bar, click the **ExeCode** button, a dialog box will pop up. In the dialog box, specify the starting and ending line numbers for the vCode to be displayed. Then click the **OK** button in the dialog box. The vCode will be displayed in a separate **Executable Code Window**.

2.3.4 Setting And Clearing Breakpoints

Breakpoints are set in the CPPE by referring to program line numbers. To set a breakpoint, from the Debugging Menu Bar, click the **Set Break** button. A **Set Break Point** window will pop up. In this window, there is a list displaying the source code starting from line number one. Setting a breakpoint by selecting a line from the source

code list, then click the **Break** button in this window. The breakpoint line will be displayed in a breakpoint list below the source code list in the same window.

If a breakpoint is set and CPPE is set to Debug mode, when users execute an application program, the execution will stop at the breakpoint and a new window titled **Step Execution From Breakpoint** will pop up showing the program source code with the breakpoint highlighted.

To clear breakpoints, from the Debugging Menu Bar, click the **Clear Break** button. The same **Set Break Point** window will pop up with the previously set breakpoints displayed in the breakpoint list. Select a breakpoint from the breakpoint list and click the button **UnBreak**.

2.3.5 Stepping Through A Process

When any running process tries to execute a line in a program with a breakpoint, the whole program execution will be suspended. At this point, the execution of the program may be continued with two functions: Continue or Step. If Continue function is used, the execution will be continued until the next breakpoint is encountered by any process. If Step function is used, the execution will be continued line by line from the breakpoint in the suspended process.

The Step function counts just executable lines in the program listing, independent of how many statements are contained on a given line. For program loops, the total number of lines executed is counted until the number reaches the number specified by the Step function. Although the Step function is applied to the currently suspended process,

all the other parallel processes also continue to execute in parallel, so they also will be advancing in execution.

To use Continue function, from the **Debugging Menu Bar**, click the button **Continue**. To use the Step function, from the **Debugging Menu Bar**, click the button **Step**. In either case, if the program execution stops at a new breakpoint of execution, the new breakpoint of source code will be highlighted in the **Step Execution From Breakpoint** window. When program execution terminates, the **Step Execution From Breakpoint** window will shut down.

To specify a different "Step-Process", from the **Debugging Menu Bar**, click the button **Set Process**, a **Set Debugging Process** dialog box will pop up. Specify the process id in the dialog box and click **OK**. To step a process with a different number of lines in each step, from the **Debugging Menu Bar**, click the button **Set Step**, a **Set Step** dialog box will be popped up. Specify the step size in terms of number of source lines in the dialog box and click **OK**

2.3.6 Tracing Variables

Whenever the execution of the parallel program is suspended, the user may want to examine the current value of variables in the current environment of each process. CPPE provides two functions for this purpose: Show and Trace.

Function Show is used to display the value of a variable when program execution is in suspension state. User can use this function to examine variables in the current environment of each active process. Active processes are those that may be in state of Ready, Running, Blocked, Delayed or Spinning.

To use function Show, the variable should be in a currently active process. To get the list of active processes, from the debugging dialog box, click the button **Status**, the default function of Status will give a full list of process status. To use Show function, click the button **Show** in the Debugging Menu Bar, a **Show Variable Value** dialog box will pop up. In this dialog box, specify a variable name and an active process id. If the variable is an array, user should specify the index range that user wants to display. Then click the **OK** button in the dialog box.

Function Trace is used to trace a particular variable during the execution process. User essentially sets a flag on the traced variable. Whenever that variable is referenced during subsequent program execution, the program will be suspended as it is for breakpoints. The traced variable may be referenced by many different processes in a variety of locations in the program.

To use function Trace, from the Debugging Menu Bar, click the button **Set Trace**. A **Set Trace Variable** dialog box will pop up. Specify the variable name and process id in the dialog box and then click button **OK**. To clear a traced variable, from the Debugging Menu Bar, click the button Clear Trace. A **Clear Trace Variable** dialog box will pop up. In this dialog box, there is a list displaying all the traced variables. User can clear a traced variable by selecting the variable from the list and then click button **OK** in the dialog box.

Users have the option to turn on or off the Trace function before or during program execution. There is a group of **Trace On/Off** radio buttons in the Debugging Menu Bar for user to turn Trace function on or off.

2.3.7 Alarm

Alarm is used to suspend the program execution when a certain amount of time is reached. The functionality of Alarm is similar to setting a breakpoint so that user can examine execution status in the process of program execution.

To set an alarm, from the debugging menu bar, click the button **Alarm**. A **Set Alarm** dialog box will pop up. User can turn the alarm function on or off from this dialog box. When the alarm is turned on, user can specify the alarm time in the text field in the dialog box.

2.4 Network Architecture And Mapping

2.4.1 Specifying The Architecture

When CPPE starts, the default architecture is a 2D-mesh parallel computer with size of each dimension being 4 (mesh 4 x 4). The user may override this default and specify a wide range of other architectures, including many of the common parallel topologies. This allows the performance of the parallel program to be simulated and evaluated on a wide range of parallel computer architectures according to the choice of the users.

CPPE has predefined some most common architectures in the system, which can be used directly by selecting from the option menu **Physical Arch** in the main frame. User can also define new architectures according to the needs of their applications. To define a new architecture, from the main menu, click the menu pane **Architecture**, then select the menu item **Add New Architecture**, a **Change Physical Architecture** dialog box will pop up. A new architecture is defined by architecture type and size. Select an

architecture type from the option menu **Topology** in the dialog box. Then specify the architecture size in the text field in the dialog box. For architecture of type Line, Ring, Fullconnect and Shared, specify the architecture size by entering the number of physical processors. For a multi-dimensional architecture, such as mesh, torus, enter the size of each dimension separated by commas. For a hypercube architecture, enter the number of dimensions. The newly defined architecture will be added to the **Physical Arch** option menu in the main frame and becomes the current parallel system architecture.

2.4.2 Virtual-to-Physical Architecture Mapping

Message-passing parallel programs are encouraged to be written using virtual topology, the topology most natural to express the program communication structure. However, the virtual topology may be the same as or different from the topology of the physical system on which the program is running. CPPE supports virtual-to-physical architecture mapping. The objectives of virtual-to-physical architecture mapping are to minimize communication cost by minimizing the distance between communicating processes, and to balance the workload among physical processors.

CPPE currently supports six types of mapping: Default, Identity, Random, Ring-to-Line, Torus-to-Mesh and User-defined Mapping. In Random mapping, virtual processors are randomly assigned to physical processors at run-time. In Identity mapping, the virtual topology are mapped to the identical physical topology, which is the exact match between virtual topology used in programming and the actual underlying physical machine. Both Random and Identity mapping can only happen when the number of virtual processors is no greater than the number of physical processors. When the number

of virtual processors is greater than the number of physical processors, effectively these two mappings will fall into Default mapping, where virtual processors are divided into blocks, and each block is then mapped to the physical processors. Ring-to-Line mapping is available when the virtual architecture is Ring, the physical architecture is Ling and the number of virtual processors equals to the number of physical processors. Torus-to-Mesh mapping is available when virtual architecture is Torus, the physical architecture is Mesh and the two architectures are identical except the wraparound links. A user-defined mapping allows unlimited mapping functions to be specified at debugging time to satisfy the need of different user applications.

To specify a virtual-to-physical mapping, select a mapping type from the option menu **Mapping** in the main frame.

To specify a user defined mapping function, users should enter a mapping function string following the syntax defined in CPPE. In general, a mapping function string has the following format:

```
mapfunction(x1, x2, ..., xn) = [y1, y2, ..., yn].
```

which maps virtual processor $(x_1, x_2, \ldots, x_n)$ to physical processor $(y_1, y_2, \ldots, y_n)$. Most C expressions can be used inside the square brackets

As an example, a one-to-one mapping from a two-dimensional torus to a two-dimensional mesh can be translated as following:

```
torus2mesh(x, y) = [x<=(#1-1)/2 ? 2*x : 2*(#1-x)-1,
                    y<=(#2-1)/2 ? 2*y : 2*(#2-y)-1].
```

where *#n* denotes the size of the n^{th} dimension of the virtual architecture, and the pair (*x*, *y*) denotes the Cartesian coordinate of the virtual processor in the virtual architecture. The square brackets enclose the right-hand side of the mapping function. The pair of values inside the square brackets denotes the Cartesian coordinate of the physical processor in the physical architecture. The final dot is used to signal the end of the mapping string so that users can break the map string in several lines during inputting.

A many–to-one mapping from a two-dimensional torus to a two-dimensional mesh can be expressed as follwoing:

```
torus2mesh(x,y)=
      [x<=(#1-1)/2 ? (2*x)/(#1/$1):(2*(#1-x)-1)/(#1/$1),
      y<=(#2-1)/2 ? (2*y)/(#2/$2):(2*(#2-y)-1)/(#2/$2)].
```

where *Sn* denotes the size of the n^{th} dimension in the physical architecture. The denominator (*#i/Si*) in the physical Cartesian coordinate provides the effect of block cyclic mapping.

To specify the user-defined mapping function, from the **Mapping** option menu in the main frame, select the **User Defined** menu item. A **User Defined Mapping** dialog box will pop up. The user can either enter a mapping function string in the text field in the dialog box, or load a mapping function string from a file that contains a mapping function string. To load a mapping function string from a file, click the button **Load** in the dialog box and a file selection box will pop up. Select the file that contains the mapping function string and click button **OK** in the file selection box. The mapping function string will be loaded into the text field in the **User Defined Mapping** dialog

box. The user can also save the user input mapping function string into a file. The button **Save** in the **User Defined Mapping** dialog box is used for this purpose.

2.4.3 Network Routing Type

CPPE can simulate different network types. Currently it supports packet switching network, simulated packet switching network, shortest path network and wormhole-routed network. In packet switching network, basic communication delay is adequate to reflect the communication delay when the message traffic is low enough that there is no interference between messages that might result in congestion delays. Some programs have more frequent communication that travel longer paths in the network, resulting in the potential of message congestion and further communication delay. In order to simulate the execution of different parallel programs, CPPE provides an option to turn on or off the message congestion. The simulated packet switching network simulates a packet switching network with message congestion turned on. The shortest path network simulates a packet switching network with message congestion turned off.

Users can select a network type before program execution starts or change network type during program execution. The network type can be selected from the option menu **Network** in the main frame.

2.4.4 Communication Delay

CPPE can simulate the parallel computer architecture, using message passing for inter-process communication. An important parameter for the inter-process communication is the Basic Communication Delay, which is the basic time to communicate a message

packet between two processors with a direct physical communication link. User can define the value of the Basic Communication Delay before program execution starts. From the main menu, click the menu pane **Architecture**, select the menu item **Network Delay**. A **Set Network Delay** dialog box will pop up for user to enter the value of Basic Communication Delay, which is defined as the number of time units.

2.4.5 Communication Parameters

In a wormhole-routed network users can check the current setting of communication parameters and redefine the values of communication parameters before simulation starts or when program execution is in suspension. The following communication parameters are defined in CPPE which are configurable during simulation: number of lane per channel, flit size (bytes), packet size (bytes), buffer size (flits), startup overhead per message (time units), startup overhead per packet (time units), headOtherFlitSpeedRatio (speed ratio between head flit and following flits). To display the current setting of communication parameters for the current routing option, from the main menu, click the menu pane **Architecture**, select the menu item **Network Parameters**, then select the cascaded menu item **Display**. The current values of the communication parameters will be displayed in the output window in the main frame. To redefine the communication parameters, from the main menu, click the menu pane **Architecture**, select the menu item **Network Parameters**, then select the cascaded menu item **Change**, a **Change Network Parameters** dialog box will pop up. The current values of all the available communication parameters are displayed in the dialog box. To change the value of a

parameter, click the corresponding text field for that parameter and reenter the value, then click button **OK** in the dialog box.

2.4.6 Vary Processor Speed

This function is used for testing multiple executions of non-deterministic applications and robustness of deterministic programs, as discussed in section 4.2.3. Race conditions are simulated by varying relative processor speeds. When the Vary Processor Speed option is turned on, users need to provide an integer Random Number Seed, which will be used to create a random number r_i between 0 and 1 (>0) for each physical processor i that will be used to increase the speed by a factor of $1/r_i$. This randomly selected speed factor for each processor will remain in effect throughout the subsequent program execution. The particular random speed factors chosen completely dependent on the Random Number Seed: using the same seed again will result in the same set of processor speed factors.

To turn on the Vary Processor Speed option, from the main menu, click the menu pane **Architecture**, then select the menu item **Vary Processor Speed**, and a **Set Vary Processor Speed** dialog box will pop up. Set the **Vary Speed** on or off from the radio selection button. When the **Vary Speed** is set on, the user can specify an integer number of Random Number Seed in the text field in the dialog box, and click button **OK**.

2.5 Program Performance Statistics

When CPPE executes a program, it keeps track of the relative timing of all processes and generates a range of performance statistics at the end of execution to help the user understand the behavior and evaluate the performance of the program.

2.5.1 Execution Time

CPPE is a code interpreter. The program source code is first compiled into a virtual-machine code (vCode) which is interpreted rather than directly executed. When CPPE runs a program, it interprets the vCode and uses an estimated execution time for each vCode instruction and keep a total execution time of the program. The estimated execution time differs between instructions depending on the complexity of the instructions. Using this estimated execution time, CPPE can simulate the performance of the program on a real multiprocessor or multicomputer.

At the end of execution, CPPE will display the total Sequential Execution Time and the total Parallel Execution Time. Sequential Execution Time is the estimated execution time on a uniprocessor computer. Parallel Execution Time is the estimated execution time on an actual target multicomputer or multiprocessor. From the ratio of sequential/parallel execution time, user can estimate the performance improvement by parallel computing.

2.5.2 Time

Time function can be used whenever program execution is suspended to give the total elapsed time since the beginning of the program execution.

To use the time function, from the debugging menu bar, click the button **Time**. The output will be displayed in the main output window in the main frame.

2.5.3 Utilization

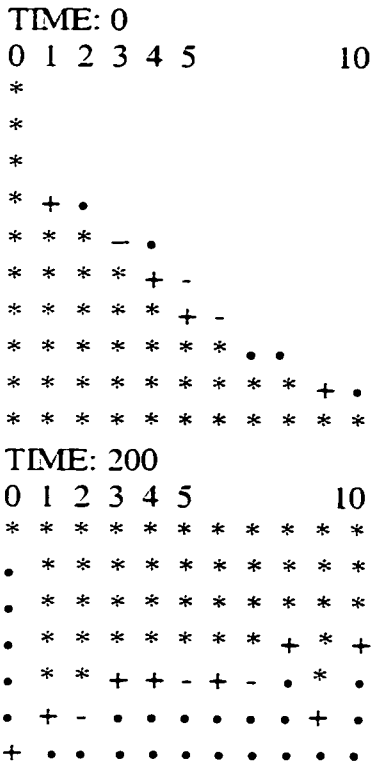
Utilization function is used to show the usage of physical processors in a particular parallel architecture. The utilization of a given physical processor is defined as the proportion of the time the processor is actually running. Whenever the program execution is suspended, the user can use this function to display the utilization of a range of processors.

To use the Utilization function, from the debugging menu bar, click the button **Utilization**. A **Process Utilization** dialog box will pop up. The user can specify the range of processors that the user wish to see the utilization value from the dialog box. The output will be displayed in the main output window in the main frame.

2.5.4 Program Performance Profile

CPPE can create a visual performance profile that will help the user to understand the program performance. The visual performance profile will be displayed in the main output window in the main frame. It shows the processor utilization during successive time intervals of program execution.

A sample profile fragment is shown as following. In the sample profile fragment, the first line shows the current elapsed program execution time. The second line shows the processor ids. Time advances vertically down the page from top to bottom, with each successive line represents a time interval of duration 10 time units. The marks indicate the processor utilization during each time interval: "*" indicates 75-100 percent utilization, "+" indicates 50-75 percent utilization, "-" indicates 25-50 percent utilization, and "." indicates 0-25 percent utilization.



To create a performance profile, the user should turn on the profile option. From the Debugging Menu Bar, set the profile option to on from the **Profile** radio buttons. The default range of processors in the profile is all processors used in the program. The default time interval in the profile is 10 time units. User can also specify a different range of processors and time interval. From the Debugging Menu Bar, click the button **Profile**, a **Show Processor Status** dialog box will pop up. The user can specify the range of processors and the time interval in the text fields provided in this dialog box, and click button **OK**.

Bibliography

[1] B. P. Lester, The art of parallel programming, Prentice Hall, 1993

[2] E. A. Brewer, 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] H. Davis, S. R. Goldschmidt, and J. Hennessy, "Multiprocessor simulation and tracing using Tango", *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991, vol.2, pp.99-107

[4] E. Reiher, H. H. J. Hum, and A. Singh, "Simulating networks of superscalar processors", *Proceedings of the Supercomputing Symposium*, 1993, pp.125-133

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

[6] B. A. Delagi, et al, "An instrumented architectural simulation system", Technical Report KSL 86-36, Knowledge System Laboratory, Stanford University, January 1987

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

[8] S. Goldschmidt and H. Davis, *Tango Introduction and Tutorial*, Computer System Laboratory, Stanford University, February 1991

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

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

[11] Dan Heller, Paula M. Ferguson, "Motif Programming Manual" for OSF/Motif Release 1.2, O'Reilly & Associates, Inc., 1994

[12] Jeff Prosise, "Programming Windows 95 with MFC", Microsoft Press, 1996

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

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

[15] L. M. Ni and P. K. Mckinley, "A survey of wormhole routing techniques in direct networks", *Computer*, 1993, pp.62-76

[16] Virginia M. Lo, Kurt Windisch, and Rajen Datta. METRICS: A Tool for the Display and Analysis of Mappings in Message-pssing Multicomputers. Proceedings of the Sixth Distributed Memory Computing Conference, April 1991

[17] Kurt Windisch, Jayne V. Miller, Virginia Lo. ProcSimity: An Experimental Tool for Processor Allocation and Scheduling in Highly Parallel Systems. Department of Computer and Information Science, University of Oregon, Eugene, OR

[18] Michael T. Heath, Jennifer A. Etheridge. Visualizing the Performance of Parallel Programs. IEEE Software, 8(5), September 1991, pp.29-39

[19] U/IMX user's manual

[20] James Gosling, Frank Yellin, and The Java Team. The Java Application Programming Interface Volume 1: Core Package: Volume 2: Window Toolkit and Applets. Sun Microsystems, June 1996

[21] F. Thomson Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann, 1991

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

[23] William A. Shay, Understanding Data Communications and Networks, PWS Publishing Company, 1995

[24] S. Chittor and R. Enbody, "Predicting the effect of mapping on the communication performance od large multicomputers", Proceedings of the International Conference on Prallel Processing, 1991, vol.2. pp.1-4

[25] Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprocessing workloads", ACM Transactions on Computer Systems, November 1988, pp.393-431

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

[27] HP-UX Symbolic Debugger User's Manual, Hewlett-Packard Company, 1992

[28] Thomas H. Cormen, charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms, the MIT Press, 1990

[29] Dimitri P. Bertsekas and John N. Tsitsiklis. Parallel and Distributed computation: Numerical Methods, Prentice Hall, 1989

[30] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karppis. Introduction to Parallel Computing: Design and Analysis of Algorithms. The Benjamin/Cummings Publishing Company, Inc., 1994

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

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

[33] C. L. Seitz, 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

[34] Advancing Programming in UNIX Environment. W. Richard Stevens, 1992

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

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