CPSS: A FLEXIBLE AND EFFICIENT SIMULATOR FOR WORMHOLE-ROUTED MULTICOMPUTERS

Hoang Uyen Trang Nguyen

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
Montréal, Québec, Canada

June 1997

0-612-40221-5

Abstract

CPSS: A Flexible and Efficient Simulator for Wormhole-Routed Multicomputers

Hoang Uyen Trang Nguyen

In this thesis, we present the design and implementation of the Concordia Parallel Systems Simulator (CPSS), a simulator for wormhole-routed multicomputers. The ultimate purpose of the CPSS is to provide a parallel programming environment which allows users to study impacts of system and software factors on program performance and to locate performance bottlenecks in parallel programs.

The major challenge in the design of the CPSS is to make a good tradeoff between the accuracy of simulation results and the feasibility of simulation time. The CPSS can accurately simulate a large range of regular topologies that represent the communication structures of most applications in scientific computations as well as the topologies of many large-scale wormhole-routed networks. Users are given the flexibility of changing communication and computation parameters as often as needed. This flexibility allows for thorough analyses of program performance under different sets of systems parameters, or on various multicomputer systems having different characteristics. The CPSS provides a rich, powerful and user-friendly set of correctness and performance debugging tools. Performance statistics at several levels of details are available for users to fine-tune their programs.

To support efficient network communication, we also propose optimal program mappings specifically designed for wormhole-routed networks. Our theoretical work in program mapping on wormhole-routed networks aims at emphasizing the importance of good mappings on such networks. As wormhole routing has become more popular, network sizes are expanded, and communication overheads are reduced, good mappings are indispensable to ensure high performance of applications running on wormhole-routed networks.

Acknowledgements

I would like to thank Dr. Lixin Tao for his thesis supervision for the last four years. My thanks also go to the CPPE team: Dr. Lixin Tao, the team leader, and Hassan Hosseini designed and implemented the CPCC; they and Thien Bui also participated in the design and implementation of the code execution module of the CPSS.

I am grateful to the professors of the Computer Science department, especially Dr. Grogono, who gave an excellent course on programming methodology (COMP244), Dr. Probst, who taught me so many courses, and Dr. H.F. Li, who has offered me guidance, advice and encouragement.

Thanks are also due to the administrative assistants of the department, especially Ms. Halina Monkiewicz and Ms. Stephanie Robert. Their friendliness and administrative support have made graduate students' life much easier. Thanks must also go to the UNIX system administrators, who have provided superior system support and services.

My good friends always deserve my appreciation: Mara Janto, Darm Muthiayen, Rong Fan and Hassan Hosseini. They have shared with me not only my accomplishments but also my frustration.

My family has been very supportive, especially my parents. I am equally grateful to my uncles and their families for their help and encouragement.

My thanks also go to my fiancé Tam for his unconditional love and support. He helped to draw most of the graphical figures in this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many scientific and engineering problems have required large-scale computations which cannot be provided by uniprocessors. Parallel processing has shown its potential to meet the demands for high computing power. People have resorted to parallel computers to speed up the processing of many important scientific and engineering applications (e.g. weather forecast, oceanography, astrophysics, computational aerodynamics, energy resources exploration, genetic engineering). To increase application performance using parallel processing is nonetheless a complex task because performance of parallel applications is influenced by many hardware and software factors such as algorithm design, system architecture, message routing technique, network speed, programming model, data and program mapping, and operating system.

In order to evaluate and improve performance of parallel applications effectively, all deciding factors must be taken into account. Also, programmers/designers should be able to observe effects of these factors on their applications so as to detect system bottlenecks and thus optimize performance of the applications (this is referred to as *performance debugging*). Unfortunately, there has been a lack of performance debugging tools for parallel applications and architectures.

The objective of this research is to provide flexible and efficient software tools for developing parallel code as well as evaluating and optimizing performance of parallel applications to be run on message-passing multicomputers.

In this chapter, we first discuss the motivations and objectives of our work. We then present the approach to realize our objectives, and a summary of our contributions. An outline of the thesis chapters is also included.

1.1 Motivations

Parallel computers are architecturally classified into two categories: shared memory and message passing. In a *shared-memory multiprocessor*, all processors uniformly share a centralized physical memory. Processes on different processors communicate via shared variables residing in the shared memory. A *message-passing multicomputer*, in contrast, has no shared memory: each processor of the multicomputer accesses its own local memory. Communication between processes is achieved by messages sent across the network connecting the processors.

Shared-memory architecture suffers from long delays of remote memory accesses. Another serious drawback of this architecture is the lack of scalability due to the centralized shared memory. It is very difficult to build a large shared-memory multiprocessor using a physically shared memory because the number of pins and pads on the memory chip is very limited. Only a few tens of processors may be connected to the physically shared memory. The shared memory can also be implemented by physically distributed memories connected by a common bus, a crossbar switch, or a multi-stage network. However, scalability is still a problem since every memory access must now go through the bus, the switch, or the network. Memory access traffic would degrade application performance significantly as the system size increases. Shared-memory architecture is thus not suitable for very large systems or applications.

Message-passing multicomputers, on the other hand, are more scalable due to the distributed nature of local memories. The main concern is latency incurred by message sends/receives. However, with the advancement of hardware technology, high-speed networks and efficient routing techniques have made message-passing architecture the developing trend for parallel computing.

This thesis studies message-passing applications and architectures. Our main goal is to help programmers write efficient parallel programs by detecting and eliminating performance bottlenecks in the programs. Performance of parallel programs depends on many hardware and software factors. Important influencing factors include system size and topology, algorithm design, program and data mapping, and routing technique.

The performance of an application may be good on systems with small sizes but degrade tremendously as system size increases. The problem here is the lack of scalability of the application. Therefore, *system size* is an important factor to be considered in evaluation of application performance.

Another factor affecting program performance is *system topology.* For the sake of programmability, message-passing programs are usually written using virtual topology, the topology most natural to express the program structure. For example, the natural topology for matrix operations is 2D-mesh or 2D-torus. The virtual topology may be the same as or different from the topology of the physical system on which the program is running. Ideally, the underlying machine topology should match the organization of the application in order to obtain best performance: mapping communicating processes to processors close to each other can help minimize communication latency. Therefore, system topology must be taken into account when selecting or developing a multicomputer for specific applications.

To achieve high performance, *algorithm design* should consider both computation and communication complexities. Algorithm design is even more crucial to parallel programs than to sequential programs due to the addition of communication cost to program performance. One of the design goals should be to minimize the amount of data to be sent. Furthermore, communicating processes should be placed close to each other to reduce communication latency and traffic. Another design issue is the scalability of algorithms. Ideally, application performance should increase linearly with the system size.

Program and *data mapping* are essential for parallel programs to achieve high performance on message-passing multicomputers. Well-designed mappings are needed to place communicating processes close to each other so as to minimize interprocessor communication cost. Data mapping is equally important. Each process must rely mainly on its own local variables and its own local portion of the shared data. The message passing for data exchange must be relatively infrequent to limit interprocessor communication.

Since communication overhead may dominate the total cost of an application, *routing techniques* have direct impacts on program performance. Whether a routing technique is suitable for a specific application depends on the communication pattern of the application. For example, in low traffic, wormhole routing offers low latency to individual messages. However, packet switching may achieve higher throughput at high loads [28].

Our objective has been to provide a parallel programming environment which allows users to study impacts of system and software factors on program performance, and locate performance bottlenecks in the program.

1.2 Approaches

We had three choices to realize our objective mentioned above:

1. Analytical modelling: Performance aspects of a parallel program under specific conditions may be estimated using mathematical formulation. This approach, however, is suitable only for simple and small computer systems and applications. Multicomputer systems and their applications are sufficiently complex to make analytical modelling very difficult.

2. Use of real machines: We could run parallel programs directly on a multicomputer to study their performance. However, testing and debugging tools supported on real multicomputers are currently very limited. Moreover, real parallel computers are non-deterministic in nature: the probability for some bugs to occur may be one over ten thousand. Thus it is very hard to test, debug and tune a program on a real multicomputer. Topologies and sizes supported by a real multicomputer are also restricted within small ranges. This limitation does not allow for studies of scalability of parallel algorithms. Furthermore, real multicomputers are expensive resources which are available to only a restricted number of users.

3. Simulation: Sequential software running on a uniprocessor can be used to emulate program execution on a real multicomputer. The uniprocessor running the simulating software is called the *host machine*; the simulated multicomputer is referred to as the *target machine*. The simulating software is supposed to accurately mimic the behavior of the target machine, and yield correct program outputs as if the program had been executed on the target system. Simulation has several advantages over the other two approaches:

    - Simulation is a more encouraging method for performance debugging than analytical modelling: it is simpler, easier to understand, and more user-friendly. In reality, systems too complex to model accurately can often be simulated, and resulting measurements can be used to guide design and performance analysis.

- The sequential simulation software is deterministic in nature. Therefore repetition of executions of a parallel program will always produce the same results under the same system parameters. This provides a stable environment to study the program at different levels of detail and from different perspectives (e.g. computation aspects versus communication aspects).
- Correctness and performance debugging is much easier with simulation due to the repeatability of sequential programs which perform simulation. (*Correctness debugging* helps to locate syntax and semantic errors in the program so as to make the program work correctly.) Simulation gives users more control over the debugging process. It permits debugging code to be inserted into simulation code: this allows users to obtain different kinds of debugging data and statistics at any given point in time. As a result, simulation helps to pin-point programming bugs, determine resource requirements, and identify system bottlenecks.
- High flexibility is provided by software systems which simulate behaviors of target systems. Simulation allows users to experiment with their programs on a wide range of topologies and system sizes (simulated system size is limited only by memory capacity of the host). This helps, for example, to study the scalability of a parallel algorithm on very large systems, which may not be feasible to do on real machines. Users are also given the flexibility to change various computation and communication parameters to tune their programs to a desired performance. Simulation also helps to predict performance of hypothetical or unavailable hardware.

Based on the pros and cons of the above three approaches, we selected the simulation approach. In this thesis, we present the design and implementation of a simulator for wormhole-routed multicomputers: the Concordia Parallel Systems Simulator (CPSS). The simulator is a sequential program written in C language and to be run on uniprocessors such as UNIX workstations and PCs. The code execution capability of processing elements and operations of the wormhole-routed network are accurately simulated. The simulator is also fast, flexible and user-friendly. To support efficient network communication, we also propose optimal program mappings specifically designed for wormhole-routed networks.

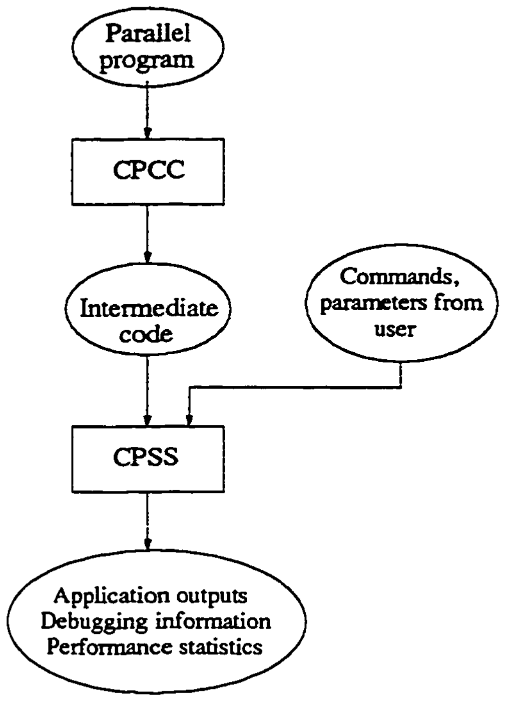


Figure 1: General structure of the CPPE

1.3 Contributions of the Thesis

The CPSS is a major component of the Concordia Parallel Programming Environment (CPPE) which consists of two main modules (Figure 1):

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

2. Concordia Parallel Systems Simulator (CPSS): The CPSS reads in the intermediate code produced by the CPCC, simulates execution of the application, yields program outputs, and provides various performance statistics. An interactive debugger is built into the CPSS to facilitate program testing and debugging.

The scope of this thesis is the design and implementation of the CPSS, which is made up of two main components:

1. Code Execution Module (CEM): The CEM plays the role of the processing elements of a multicomputer system; it executes the intermediate code produced

by the CPCC.

2. Network Manager: The role of the network manager is to allocate network resources to messages, route and deliver messages, and detect deadlock in the network if any.

The thesis also presents optimal mappings for programs running on wormhole-routed networks. Optimality of proposed mappings are proved theoretically and confirmed by experimental results obtained from the simulator.

1.3.1 Parallel System Simulator

Two potential problems simulation usually encounters are accuracy and speed. There is a tradeoff between these two factors: the more accurate the simulation is, the longer the simulation time takes in general. Our simulator has been carefully designed to give most accurate simulation results within a reasonable amount of time.

The CPSS uses the functional simulation technique [13, 34, 25] which offers the most accurate results among the existing simulation techniques (e.g. trace-driven [10, 11, 21], direct execution [32], code-augmented direct execution [12, 19, 35]). This technique interprets parallel object code instructions at the functional level, hence the accuracy. In the CPSS, accurate simulation is also achieved by parameterizing system measurements (e.g, system clock cycle, execution times of object code instructions of the target architecture, packet size, link buffer size, link delay, message and packet startup overheads). The ability of changing system parameters allows the user to simulate any target architecture accurately by just redefining the system parameters in the simulator to reflect the characteristics of that architecture.

As far as performance is concerned, parallel primitives are interpreted at a reasonable level of abstraction so as to obtain fast simulation and not to compromise simulation accuracy. We do not go into very low levels of details but retain essential characteristics of the target processors and network. The entire simulation system, including the application program, is run by a single process. The simulation does not incur any host context switching, and thus saves simulation time.

Besides accuracy, the functional simulation technique also outperforms the other simulation techniques in terms of flexibility and convenience of debugging. Our simulator can simulate a wide range of multicomputer topologies and sizes. It also supports a large set of configurable parameters which permit users to fine-tune their applications and simulate various multicomputer systems. Moreover, the same parallel program can be mapped to different physical architectures at run time. In the CPSS, parameter or mapping changes do not require any modifications to the simulator program or the application program. This convenience is unique to the CPSS among the existing multicomputer simulators. In other simulators [12, 19, 35], when a parameter (e.g. physical topology) is changed, the simulator program must be modified, re-compiled and re-linked with the application.

The routing technique currently supported by the CPSS is wormhole routing. The design and implementation of the simulator are modular and decoupled. Therefore, any kind of network other than wormhole routing (e.g. packet switching, circuit switching) can be implemented independently and integrated into the simulator easily.

The CPSS provides many debugging tools to facilitate users' code development. Design concepts of the debugging tools are borrowed from sequential programming environments to make the tools as user-friendly as possible. Performance statistics at various levels of details are also available to support algorithmic and architectural performance evaluation and tuning.

Finally, the CPSS provides repeatability which is essential for implementing a stable and reliable debugging environment. On real machines, a deterministic application may generate a different result for each run due to race conditions and varied speed of CPUs and routers. The CPSS also supports multiple executions of a non-deterministic application to give users a broad and accurate view of the application's behaviors under different outcomes of race conditions. Multiple executions are equally useful for testing the robustness of a deterministic application.

1.3.2 Wormhole-Routed Network Simulator

This is a concise, accurate, fast and flexible wormhole-routed network simulator.

Messages, packets, flits, physical links, virtual channels, arbitration queues, and operations on these entities are simulated in details. Simulation results are thus very accurate. To speed up the simulation process, we selectively left out low-level hardware details which do not compromise simulation results. Also, to reduce simulation time, we employ an approximate version of round-robin scheduling of virtual channels time-sharing a physical link. Default arbitration/allocation schemes are either FIFO or round-robin (wherever applicable) to obtain fast simulation. However, arbitration/allocation schemes are parameterized so that other schemes can easily replace default settings.

The network simulator can simulate a wide range of topologies (lines, rings, meshes, tori and hypercubes), and accommodate large networks of up to thousands of nodes. Most network parameters are configurable to allow users to fine-tune performance of their programs as needed. The simulator can be used as a stand-alone network simulator or embedded into the CPSS to simulate execution of real applications.

1.3.3 Optimal Mappings on Wormhole-Routed Networks

Unlike packet switching, wormhole routing has communication latencies that are nearly independent of path lengths [17, 8]. Therefore minimizing path lengths is no longer a mapping objective for wormhole-routed networks [1, 3, 17]. Random mapping is claimed to be good enough for second generation wormhole-routed multicomputer systems [9]. In fact, small system sizes and high communication overheads are limiting the effect of contention from becoming serious in current systems. With an increase in system sizes and a reduction in communication overheads in future systems, contention can significantly degrade communication performance [2, 4]. Nevertheless, there has not been enough emphasis on the importance of program mapping on wormhole-routed networks or on good mappings themselves.

In this thesis, we propose several optimal mappings for programs running on wormhole-routed networks. The mapping objective is to minimize link contention among messages. The mappings cover a wide range of topologies and system sizes. The optimality of the mappings is proved theoretically and then justified by experimental results obtained from real parallel applications running on our simulator. Simulation results show that our mapping functions significantly outperform random mappings in terms of communication performance, especially on large networks.

1.3.4 Summary

The CPSS is especially useful for the development of architecture-independent parallel applications. It is intended to be a tool for evaluating impacts of system and software factors on application performance. The ultimate goal is to determine performance bottlenecks in the program so as to improve its performance. The simulator is equally helpful to the design and analysis of novel multicomputer systems. It can also serve as a teaching tool for users who want to learn parallel programming. The simulator is accurate, fast, and flexible; its debugging environment is user-friendly.

Our theoretical work in program mapping on wormhole-routed networks aims at stressing the importance of good mappings on such networks. As wormhole routing has become more popular, network sizes are expanded, and communication overheads are reduced, good mappings are indispensable to ensure high performance of applications running on wormhole-routed networks.

1.4 Thesis Outline

In chapter 2, we present a critical review of existing multicomputer simulators and wormhole-routed network simulators. The review includes an analytical comparison of different simulation techniques. Chapter 3 gives an overview description of the CPSS. Design objectives and system modelling are discussed. Major components of the CPSS are also described at a high level. In chapter 4, the design and implementation of the code execution module are described in details. The description focuses on parallel execution aspects of multicomputer systems, and highlights the simulation techniques we have used. We then discuss the design and implementation of the wormhole-routed network simulator in chapter 5. In chapter 6, we propose several optimal mappings for programs running on wormhole-routed networks. The optimality of the mappings is theoretically proved, and then confirmed by experimental results obtained from the simulator. Chapter 7 provides a summary of the thesis and suggestions for future work.

Three additional appendices are given at the end of the thesis. Appendix A provides a description of parallel features of the CPC language which allow the creation of parallel processes, the definition of parallel architectures, process communication via channel variables, and process-to-processor mapping. Appendix B serves as a user's manual for using the CPSS. It begins with general instructions on how to use the CPSS to run programs. Then each command of the interactive debugging tools is described. Finally, the set of intermediate code instructions used by the CPPE is listed in Appendix C.

Chapter 2

Literature Survey

The general structure of a multicomputer system consists of a set of processors (or nodes) connected by an interconnection network as illustrated in Figure 2. Each processor has its own processing element (PE) , local memory (LM), router (R) and other supporting devices. The processing element and the local memory run computation activities, while the router and the network support communication among processors via message passing.

Consequently, simulating a multicomputer system should effectively reflect two separate aspects of the system: computation and communication. A good simulator should accurately simulate both computation and communication activities taking place in the target system. However, depending on the purpose of a simulator, the simulation of either aspect can be simplified to focus on only one kind of activity. For example, the Multi-Pascal simulator [34, 25] emphasizes on the accuracy of code execution simulation (i.e. computation aspect) but has a very simple implementation for network communication. In fact, the simulator does not support dynamic network simulation at all; arrival times of messages are calculated based on a predetermined communication model. The PARSE simulator [26], in contrast, simulates network activities in details but simplifies code execution simulation for more efficiency.

Our simulator is intended to provide accurate simulation of both computation and communication activities. The purpose of the simulator is to provide a flexible and user-friendly environment for correctness and performance debugging of parallel programs. We also focus on the issue of program mapping on wormhole-routed networks to help programmers further optimize performance of their applications. Another important objective is to produce simulation results within acceptable time limits.

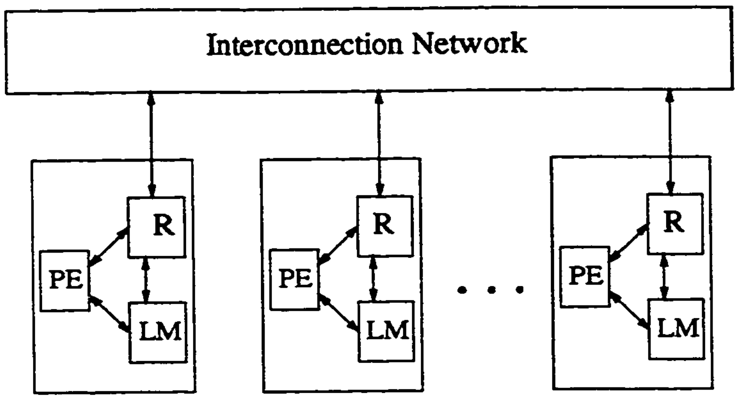


Figure 2: Modeling the multicomputer system

This chapter presents a literature review of existing code execution simulation techniques and typical simulators which employ these techniques. Wormhole-routed network simulation techniques are also discussed in a similar manner. We also provide a literature survey on program mapping on wormhole-routed networks.

2.1 Code Execution Simulation

Several simulation systems for parallel computers have been developed [12, 19, 35, 25, 26]. Existing simulation techniques can be classified into three categories.

1. Direct execution. A parallel program is first compiled into object code which is in the assembly language of the host. During compilation, the compiler identifies two kinds of instruction for the purpose of simulation: local instructions and non-local instructions. An instruction is local if it has effects on only the local processor. Examples of local instructions are register-to-register instructions or memory accesses to a local variable residing in the local memory. Non-local instructions, in contrast, impact another part of the system such as a remote processor or the network. In particular, non-local instructions perform parallel tasks such as process creation/termination, message sends/receives or process synchronization. Each non-local instruction will be simulated via a procedure call which interprets the instruction at the functional level. Local instructions, on the other hand, are executed directly by host processes and timed with the host's clock. This simulation technique is fast but not accurate

since the simulation is timed with the host's clock and not the clock of the target architecture.

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 count of an instruction is the time it would take to execute this instruction on the real machine. 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 approach thus results in a more accurate simulation than the pure direct execution technique.

3. Functional simulation. A parallel program is first translated into intermediate code of a virtual parallel machine. The set of intermediate code instructions is definable and can be different from the host's assembly language. 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 in general takes more simulating time than the other two techniques, but its simulation results are the most accurate.

The following sub-sections will analyze these techniques in details. We will discuss their characteristics, advantages, drawbacks, and example simulators.

2.1.1 Direct Execution

In this approach, a parallel program is first compiled into object code which is in the assembly language of the host. During compilation, non-local instructions are converted to procedure calls that interpret the instructions. Local instructions are compiled directly into host assembly instructions. During program execution, each application process is simulated by a distinct host process. Local instructions of an application process are executed directly by the corresponding host process since they do not affect other parts of the system. Non-local instructions, however, need to be interpreted at the functional level. That is, the behavior of each non-local instruction is emulated by a host routine as if the instruction were being executed on the real system. There is an additional process (called the *simulation engine*) which executes all non-local instructions. Since non-local instructions affect components other than their local processors, their execution must be centralized and coordinated to ensure program correctness. When an application process encounters a non-local instruction, the corresponding host process passes the control to the simulation engine which will run that non-local instruction.

An example simulator is the CARE simulator [32, 33] which simulates LISP code using direct execution and a hardware timer. In this simulator, non-local instructions are interpreted by the simulation engine. Local instructions are directly run by the host's simulating processes and timed with the host's clock.

This technique is generally faster than the functional simulation approach because local instructions are executed directly instead of being interpreted. However it suffers from a major drawback: difficult debugging. Local instructions are directly executed by the host and the simulation engine does not have much control on the execution of local instructions. Thus it is very difficult to establish the connection between user-application statements and low-level data or simulation activities. Such connection is essential for in-session debugging and fine-tuning an application. (In-session debugging refers to the debugging interaction between the user and the program *during* execution of the program. Examples of in-session debugging are single-stepping the program, setting break points or tracing a variable after breakpoints). For example, it is very hard to examine the value of a local variable belonging to a particular process. Sequential debugging tools such as *dbx* are not able to locate the desired process in order to access its local variable (unless monitoring code is added to the application to identify the desired process). In-session debugging is in general not feasible with the direct execution technique.

Because each application process is simulated by a host process, context switching will incur high overhead if the number of application processes is considerable. Furthermore, the number of application processes that can be simulated concurrently is very limited.

An even more serious drawback of this approach is low accuracy. The reasons for inaccurate simulation results are:

- Coarse granularity of the host's clock which is usually workstation clocks. Therefore, the timing is not accurate because execution time of an instruction is often truncated to the nearest milliseconds. Within a millisecond, the target multicomputer may have executed thousands of instructions or sent hundreds of messages.

- Control code added to monitor the simulation. Monitoring code is often needed because in-session debugging is very hard with direct execution. Such code fragments would not be executed on the target machine. However, in this approach, there is no way to distinguish the monitoring code from the application code. Therefore the direct execution technique also times the monitoring code and this affects the overall execution time.

In summary, the direct execution technique is fast but inaccurate. This approach is usually used for studying hardware features such as caching in shared-memory multiprocessors or network performance evaluation. In such applications, the accuracy of code execution simulation is not important. In addition, debugging tools provided by direct-execution simulators are very limited and based primarily on monitoring code added to the application and the simulation engine.

This approach is not suitable for the purpose of our simulator, which is to accurately simulate both computation and communication activities of a target multicomputer, and to provide a user-friendly debugging environment.

There exists an enhancement to the direct execution technique, which employs code augmentation to count execution time of local instructions on the target system.

2.1.2 Direct Execution with Code Augmentation

Code augmentation is an extra step added to the compilation process, which inserts cycle counts to the compiled object code. The cycle count 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 multicomputer. This results in a more accurate simulation.

Like the pure direct execution technique, code-augmented direct execution is generally faster than functional simulation since local instructions are not interpreted but executed directly. Code augmentation, on the other hand, 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 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 instructions blocks) can be problematic. A simple addition will change the behavior of the simulation since the cost of monitoring code is also included in the cycle counting. Conditional compilation flags or macros can be used to exclude the cost of added monitoring code [12]. 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 monitoring code). This would increase the cost and thus could change the behavior of the system. The more debugging or statistic traces are required, the more perturbed the simulation can be.

Several simulators were implemented using the code augmentation approach. Typical simulators of this kind are Proteus [12], Tango [19, 22], EPPP [35, 36] and PARSE [26].

Proteus

The pure direct execution technique correctly simulates the functionality of local instructions but ignores the exact calculation of the actual execution time. Proteus [12] (developed at MIT in 1991) uses code augmentation to count the cycles required by the target machine to execute local instructions.

The application program is first compiled into the host's assembly language. 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.

Each application process of the target is simulated by a light-weight process (thread) of the host. Context switching on the host is required to interleave execution of threads which run local instruction blocks. Each thread context switching takes 3 microseconds. Non-local instructions are implemented by procedures and interpreted by the simulation engine as in the pure direct execution approach. During program execution, the simulation engine also manages simulating threads: when a simulating thread finishes execution of a basic block, it updates the cycle counter of its simulated processor and gives control to the engine. The engine then selects the next available block for execution and passes control to the corresponding simulating thread. The engine also handles interprocessor communications.

A specific engine must be defined for each simulated MIMD architecture. When the user chooses to simulate a specific multicomputer system, the user has to modify the parameters of the engine. The engine is then re-compiled and linked with the user's application.

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, an graph generator is used to interpret the trace file data and present the results of the simulation.

Although Proteus simulation is fast, it suffers from several drawbacks.

- The timing results may not be accurate because the cost of each basic block is determined at compile time and is a fixed number. In reality, the cost of an instruction depends on other run-time factors such as the operands (or cache hits if the target machine is a shared-memory architecture).

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

- Simulation performance may be substantially degraded if the application involves many processes. In addition to the simulation engine process, a host process is created for each application process. If the number of application processes is large, this may incur substantial overheads of context switching among the simulating threads. In practice, augmentation overhead is an insignificant part of simulation cost. Simulating non-local instructions and context switching dominate the cost of simulation [12].

- The simulator is not flexible from the user's point of view. When the architecture is changed, the engine parameters must be modified. The engine is then re-compiled and linked with the user application. This is not convenient, for example, for experimenting with program mappings. This 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 [19, 22] was built at Stanford University in 1990. Tango and Proteus were developed independently but they are quite similar. However Tango simulates only shared-memory architectures.

Application programs are written in C or Fortran. Parallel features are provided by macros. For instance, *Lock* acquires a binary lock and *Unlock* releases it. The compilation process consists of five steps: macro expansion, compilation into assembly language, code augmentation, assembly and linkage. If a parameter needs to be changed, the simulation engine must be modified and the compilation process is repeated.

Like Proteus, Tango may produce inaccurate simulation results due to fixed costs of local instruction blocks calculated at compile time. Similarly, the target system is assumed to have a basic instruction set that can be approximated by the host architecture in order to obtain accurate simulation.

Novel target machine instructions that do not exist on the host are implemented in libraries and macro packages and will be interpreted at run time. However, if the target machine instruction set differs considerably from the host instruction set, the simulation would approach functional simulation.

Tango's performance is not as good as that of Proteus. Tango uses Unix processes to simulate parallel execution while Proteus uses faster light-weight processes managed by the simulation engine. Context switching time in Tango is 180 to 250 microseconds [19]. If the application execution involves a large number of processes, context switching cost is significant.

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 process summary file and event trace file. This is not a user-friendly debugging environment for parallel applications.

Tango was implemented for studying shared-memory behaviors, shared-memory synchronization and concurrency abstractions, and for architectural evaluation [19]. It can also be used for application studies. However debugging tools are not adequately provided for code development or performance fine-tuning.

EPPP Project

The EPPP (Environment for Portable Parallel Programming) simulator [35, 36] is in fact an extended version of Proteus. Target architectures of the EPPP simulator are superscalar/superpipelined processors. Applications are written in C with extended parallel features for superscalar/superpipelined operations.

In this simulator, the augmentation phase is enhanced to accurately simulate a particular target architecture whose instruction set differ 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 [35].

Like Tango, no in-session debugging tools are available in the EPPP. The only available debugging feature is the optional generation of an extended version of PICL traces [37]. Traces are then analyzed and interpreted by a software.

PARSE

Unlike Proteus or Tango which uses a separate program to augment the compiled code, PARSE [26] 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.

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 (pure or with code augmentation) is fast but always 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 monitoring code perturbs system and application behaviors. The more traces/data required, the less accurate simulation results. This approach is thus not well suited 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.

Direct-execution simulation is usually employed for studying hardware-related features such as caching, shared memory behaviors, or network properties where accuracy of code execution is not a critical objective.

To effectively support code development and application performance fine-tuning, another simulation technique is required, which is more accurate and can accommodate more powerful debugging tools.

2.1.3 Functional Simulation

This technique interprets instructions of the target machine at the functional block level as if they were being executed on the target. Each instruction of the target machine is usually expressed as a host macro/procedure whose size depends on the complexity of the instruction and the desired level of simulation accuracy.

A parallel program is first translated into intermediate object code. The set of intermediate code instructions can differ from the host's assembly language. Ideally, it should be defined to match the target instruction set. However this is not a necessary condition for accurate simulation because the simulator designer has control over how to interpret intermediate instructions. The designer may approximate a target instruction by modifying interpretation code of the corresponding intermediate instruction.

The level of detail of interpreting intermediate code determines simulation accuracy and time. There is a tradeoff between these two factors: the finer granularity of interpretation, the more accurate results and the longer simulation time. The finest level of interpretation is machine-instruction level which is architecture-dependent and very time-consuming.

Although functional simulation generally takes more simulation time than the direct execution approach, it is a very attractive technique for performance debugging due to

- Very high accuracy. This is due to the interpretation of intermediate instructions as if they were executed on the target machine. Also, intermediate instructions can be added or removed from the instruction set, and re-defined in terms of its functionality to match the target instruction set. In addition, cycle counts of intermediate instructions are accumulated at actual run time (whereas direct-execution simulators compute execution time of local blocks in advance, at compile time).

- Flexibility. The set of intermediate instructions can be modified and functionality of instructions can be re-programmed to match the target architecture. Also, the level of accuracy can be traded for faster simulation by adjusting the detail level of code interpretation.

- Flexible and convenient debugging. Code interpretation permits the simulator to have complete control over program execution. This allows to establish the connection between user program statements and intermediate instructions. The user can thus set breakpoints, examine trace variables, or single-step the program fragment belonging to 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 at all: execution time of monitoring code is not accumulated.

A typical simulator employing this technique is Multi-Pascal simulator [34, 25].

Multi-Pascal Simulator

This simulator simulates both shared-memory and message-passing architectures. The programming language for writing applications is Multi-Pascal [25], which is based on Pascal and added with parallel features to express parallel operations such as process creation/termination, message send/receive and process-to-processor mapping. Parallel operations are defined at a high-level of abstraction to enhance programmability. For instance, message send and receive are expressed in terms of ordinary Pascal assignments. Writing to a channel variable represents a message send; reading from the channel variable corresponds to a message receive [25].

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 multicomputer. At run time, intermediate code instructions are interpreted and their cycle counts are accumulated properly.

The simulator and the application are run by a single host process. Parallelism is simulated by time slicing: each application process is given a quantum to run and application processes are scheduled in a round-robin fashion. When every application process has finished its quantum, the global clock is advanced to the next quantum.

The implementation is similar to round-robin scheduling of processes on a multitasking uniprocessor, except that the global clock is not accumulating running times of all processes but is incremented by the duration of the quantum. Since all application processes are simulated by a single host process using time slicing, there is no context switching overhead on the host.

Multi-Pascal simulator provides a rich set of debugging tools. It allows users to set instruction breakpoints and time breakpoints, define and examine trace variables, and single-step the code of a specific process. Users can also view status of processes and processors after a breakpoint, processor utilization as a function of time, and computation/communication time. Multi-Pascal debugging environment is a good illustration of flexible and convenient debugging provided by the functional simulation approach.

Despite the advantages inherited from the functional simulation technique, Multi-Pascal still has some limitations which prevent it from being a performance debugger.

- Cycle counts of intermediate instructions are hardcoded into the interpreting code of the instructions and not well-defined. For instance, an integer operation takes the same amount of time as a floating point operation, which is one time unit. Simulation results thus may not be accurate. In fact, the intended use of Multi-Pascal simulator is as a parallel programming teaching tool for students and novice programmers. It was not meant to be a tool for studying performance of real parallel applications.

- The simulator does not support the concept of virtual architecture. The architecture declared in the application program is also the physical architecture. Users need to specify process-to-processor mapping in the application (unless they wish to use the default mapping provided by the Multi-Pascal compiler); there is no run-time mapping. If the physical architecture or the mapping is changed, the application needs to be modified and re-compiled. This limitation makes study of program and data mapping inconvenient and time-consuming. 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.

- The simulator assumes an underlying packet-switching network. However, there is no dynamic network simulation. Communication overheads of message sends and receives are calculated based on a communication model.

- The simulator does not support file I/O. In fact, it is not intended for large applications.

Our simulator (CPSS) attempts to overcome the above limitations to offer parallel programmers an accurate, fast, flexible and user-friendly performance debugger.

2.1.4 CPSS Simulation Technique

The CPSS uses the functional simulation approach to simulate execution of parallel programs on a multicomputer system. Applications are written in the CPC language which enhances the C language with parallel features to express process creation/termination and message sends/receives. Parallel operations are defined at a high level of abstraction and reuse existing syntax of the C language wherever possible to promote programmability and ease of learning.

The intermediate instruction set is designed based on an analysis of common operations of parallel systems. The objective is to simulate a wide range of message-passing multicomputers. Every intermediate code instruction is associated with a configurable cost which can be adjusted to match a specific target.

The intermediate instruction set can be extended if it is different from the target's instruction set. The implementation of the simulator is modular and decoupled. So a new intermediate instruction can be added easily to the simulator as a routine which interprets the instruction. The addition of a new intermediate instruction does not affect the simulation of another target whose instruction set does not contain the new instruction since this target would not use the added routine at all.

One of our design goals is to obtain simulation outcomes fast. We do not go into low level details but maintain the essential characteristics of instruction behaviors on target machines in order to yield program outputs within reasonable time limits. Also, the intermediate instruction set is defined at a high-level of abstraction. Application processes are run by a single host process. So there is no host context switching during simulation. This helps to simulate applications with large numbers of processes efficiently.

The CPSS supports virtual architecture programming and run-time mapping to improve programmability of message-passing applications. The burden of program mapping is now shifted from the user to the simulator. 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. Moreover, the same source program can be mapped to different physical architectures without any changes to the source code. The simulator provides a library of optimal and optimized mappings.

The CPSS contains a dynamic network simulator. The simulated network is wormhole-routed, flit-based and time-driven. Packets are routed link by link until completely received. The network simulator offers very accurate message routing and communication performance statistics.

Similar to Multi-Pascal simulator, the CPSS provides users with a rich set of debugging tools. Users can set instruction and time breakpoints, define trace variables and single-step the source code of a particular process. As a performance debugger, the simulator allows users to define system parameters, examine status of processes, processors and messages, and view computation and communication statistics.

The CPSS is also very flexible and convenient. Users can configure most computation and communication parameters. Values of the parameters can be changed within the same simulation session as often as needed. No re-compilation is required: the same intermediate code of the application and the same simulator code are always executed. This flexibility is unique to CPSS among the existing multicomputer simulators.

2.2 Wormhole-Routed Network Simulation

In this section, we first review existing routing techniques to highlight the advantages of wormhole routing over the other techniques. Example wormhole-routed network simulators and their simulation techniques are then discussed. We also provide a brief description of our network simulator, its simulation technique and characteristics.

2.2.1 Routing Techniques

This subsection provides a concise description of existing routing techniques and their characteristics. These techniques are packet switching, virtual cut-through, circuit switching, and wormhole routing.

Packet Switching

This technique is also called store-and-forward switching [23, 8, 47]. In a packet-switched network, when a packet arrives at an intermediate node, the entire packet is stored in a packet buffer. The packet is then forwarded to the next node on the path when the next output channel is available and the next node has an available buffer.

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

Packet switching is simple to implement. It was used in first-generation commercial multicomputers such as Intel iPSC/1 [43], nCUBE/1 [44], Ametek S/14, and FPS T-series [45]. However, it has become less popular because of the following serious drawbacks.

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

Virtual Cut-Through

This approach can be considered as an enhancement to the packet switching technique [46]. The enhancement is to reduce the amount of time spent transmitting data. A packet is buffered at an intermediate node only if the next required channel is busy.

The network latency for virtual cut-through is $(H/B)D + P/B$ where H is the length (in bits) of the packet header that stores control information such as the destination address and the packet sequence number. If $H << P$, the second term, P/B, will dominate the total latency. In this case, the distance D has a negligible effect on the communication latency.

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

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

Circuit Switching

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

1. Circuit establishment phase: a physical circuit is constructed between the source and destination nodes.

2. Packet transmission phase: the packet is transmitted along the established circuit to the destination. During this phase, the channels constituting the circuit are reserved exclusively for this packet. Thus no buffering is needed at intermediate nodes.

3. Circuit termination phase: the circuit is released as the tail of the packet is transmitted.

The network latency for circuit switching is $(C/B)D + P/B$ where C is the length (in bits) of the control data transmitted to establish the circuit. When $C << P$, the distance D will produce a negligible effect on the network latency.

Second-generation multicomputers such as iPSC/2 and iPSC/860 [53] employ circuit switching because of its lower network latency and reduced buffer space requirements. However, it is very difficult for circuit switching to support sharing of physical links among contending packets. This results in low network utilization and susceptibility to deadlock.

Wormhole Routing

In a wormhole-routed network, a packet is divided into a number of flits (flow control digits) for transmission. Unlike the other routing techniques, wormhole routing requires the buffer at each node to store only a few flits.

As soon as a node examines the header flit of a packet, it selects the next link on the route and begins forwarding flits down that link. The header flit governs the route. As the header advances along the specified path, the remaining flits follow in a pipeline fashion. If the header flit encounters a link in use, it is blocked until the link is freed. The flow control also blocks the following flits and they remain in flit buffers along the established path. When the last flit of the packet (the tail flit) leaves a node, the link assigned to that packet is released and may be reassigned to another packet.

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

Virtual channels time-sharing a physical link are possible in wormhole-routed networks. However, virtual channels occupied by a message are not relinquished even if the message is currently blocked, and this may lead to deadlock. Deadlock-free routing algorithms [15] for wormhole-routed networks have been proposed, which multiplex multiple virtual channels on the physical link, properly number virtual channels and route messages in order of decreasing (or increasing) of virtual channel numbers.

Many multicomputer systems have used wormhole routing; among them are Ametek 2010 [9], nCUBE 6400 [48], Intel/DARPA's Touchtone Delta, Intel Paragon [49], Intel/CMU's iWarp [50], and the Transputer IMS T9000 family [51]. The popularity of wormhole routing is due to the following advantages.

- Network latency is relatively insensitive to path length.
- Required buffer space is small: only small FIFO flit buffers are required instead of large buffers for packets.
- It is easy to support virtual channels which allow contending packets to time-share a physical link. Virtual channels help to improve network utilization, and are used to implement deadlock-free routing algorithms in wormhole-routed networks.
- Wormhole routing allows packet replication in which copies of a flit can be sent on multiple output channels (virtual channels). Packet replication is useful in supporting broadcast and multicast communication [38]. Circuit switching, by its nature, does not allow packet replication.

Summary

Virtual cut-through, circuit switching and wormhole routing are more attractive than packet switching due to lower latency and reduced buffer space requirements. However, the behavior of virtual cut-through will resemble that of packet switching if network contention is significant. Circuit switching, on the other hand, is susceptible to deadlock and may result in low network utilization due to the absence of virtual channels.

Wormhole routing is currently the most popular routing technique because of low network latency, small buffer requirements and easy support for virtual channels. Several wormhole-routed network simulators have been implemented for the purpose of studying/evaluating wormhole routing characteristics and network performance [2, 29, 18]. In the following subsection, we describe typical simulators and their simulation techniques, useful features, and drawbacks

2.2.2 Wormhole-Routed Network Simulators

All existing wormhole-routed network simulators employ the time-driven approach instead of the commonly used discrete-event simulation approach because it is faster, simpler to implement, easier to understand, and yet very accurate.

Chittor's Simulator

This simulator [2] was implemented to study the effects of random and optimized mappings on communication performance.

The simulator is very simple. Messages are randomly generated and not packetized; each message is treated as one packet of the same size. Flit size is also a fixed number (8 bits). The simulator does not support virtual channels. The buffer size at each link is one flit (8 bits). Therefore the simulator can simulate networks of at most 256 nodes. The following topologies are available: line, hypercube, 2D-mesh, 3D-mesh and tree. However, every time the topology is changed, the simulator needs to be re-compiled.

The simulation is time-driven. There is a global clock. As the clock advances by one time unit, messages that are not blocked move forward by one link. Routing algorithms are deterministic (e.g. XY-routing for meshes and E-cube routing for hypercubes). Message routing is simple since the simulator does not implement virtual channels. As a message moves forward, only the positions of the head flit and tail flit need to be recorded.

In summary, this simulator is very simple. It simulates the very basic wormhole-routed network with no variations or enhancements (e.g., virtual channels, packetization, configurable parameters). The major drawback of the simulator is the absence of virtual channels which are crucial for implementing deadlock-free routing algorithms in wormhole-routed networks [15].

Saha's Simulator

This simulator [29] was designed to study wormhole-routed networks embedded within real-time, application-specific multicomputers. The network of such systems must also meet stringent timing and dependability requirements of real-time systems.

The simulator simulates only one topology which is 2D-torus and employs deterministic routing (XY-routing).

The simulation is time-driven and node-driven: for each cycle of simulated time, every node in the network is simulated. Messages of arbitrary length are created and divided into flits. Each node injects a message with a pre-determined probability. The destination address of the message is randomly generated with a uniform distribution. The message is assigned a priority. The router will use message priorities for all arbitration/allocation decisions wherever applicable.

Each arbitration/allocation scheme is implemented by a prioritized queue based on message priorities. Thus the simulation is slow due to management of different kinds of prioritized queues in the system.

Virtual channels are implemented in this simulator. Virtual channels of each link are organized in the form of a prioritized queue.

The simulator is flit-based: it deals with individual flits. As the header flit of a message is advanced, the non-header flits of the message simply follow the established path if possible.

In summary, the simulator is time-driven, node-driven and flit-based. Virtual channels are effectively simulated. However, the simulation is slow, especially for large networks, due to management of different kinds of prioritized queues. All arbitration/allocation decisions are based on priorities of messages. This is a stand-alone network simulator with no program execution component. Messages are randomly generated.

Dally's Simulator

This simulator [18] supports k-ary n-cube and k-ary n-fly networks. It simulates interconnection networks at the flit-level. A flit transfer between two nodes is assumed to take place in one time unit. The network is simulated synchronously moving all flits that have been granted channels in one time step and then advancing time to the next step.

For a specific network topology and size, users can define the number of virtual channels per link and the routing algorithm (deterministic or adaptive). The simulator provides two algorithms for link bandwidth allocation: random and the oldest-packet-first scheme (deadline scheduling). Deadline scheduling is claimed to help reduce average message latency and make message latency more predictable [18].

Messages are randomly generated using a pre-determined probability depending on the purpose of a specific experiment. Each message is also a packet of fixed length (20 flits). Flit size is not considered in this simulator. Operations of virtual channels and routing algorithms are simulated in details.

In summary, the simulator is time-driven and flit-based. It simulates virtual channels and routing algorithms in details. The simulation is slow due to many low-level hardware details and too many phases involved in a communication step. This is also a stand-alone network simulator with no support for running real applications. In fact, it was implemented to study the effect of virtual channels on network performance (throughput and latency) [18]. Although the simulator is very accurate, its simulation time would be unreasonable for our purpose.

2.2.3 Our Network Simulation Technique

Our network simulator also employs the time-driven technique. The simulator is time-driven and message-driven: for every network cycle, the network manager attempts to advance each flit of every packet by one link if possible. Physical links are scheduled for virtual channel time sharing only if required (i.e. if a link is used by at least one packet); unused links are not scheduled. We use an approximative version of round-robin scheduling to speed up the simulation time. The scheduling algorithm will be described in details in section 5.4.3.

To further reduce simulation time, we selectively left out low-level hardware details. For instance, the network simulator does not route actual contents of messages but simulates the routing using sequence numbers of packets and flits. Therefore flit buffers need not be implemented. Furthermore, default arbitration/allocation schemes are either FIFO or round-robin (wherever applicable) for fast simulation. However, arbitration/allocation schemes are parameterized so that other schemes (e.g. prioritized queues) can easily replace default settings.

The network simulator is very flexible. It can be used as a stand-alone network simulator or embedded into the CPSS to simulate execution of real applications. A wide range of topologies is supported. The simulator can accommodate large networks of up to thousands of nodes (subject to the memory capacity of the host). Moreover, users can define most network parameters (packet size, flit size, virtual channel buffer size, routing scheme, link bandwidth, number of virtual channels per link, virtual channel allocation scheme, link scheduling algorithm, etc.).

2.3 Program Mapping on Wormhole-Routed Networks

Second generation multicomputers (iPCS/2, Symult 2010, iWarp, nCUBE/2) use wormhole routing or circuit switching instead of packet-switching employed in the first generation multicomputers (iPSC/1, nCUBE/1). In packet-switched networks, communication latency is sensitive to the distance between the source and destination nodes. As a result, program mapping on packet-switched networks has always aimed at minimizing *dilation costs* [6, 7]. Dilation cost of a mapping is the maximum distance between any two communicating processes.

Wormhole routing, in contrast, offers low network latency that is relatively independent of path length. In fact, experimental results confirm that in wormhole-routed networks communication performance does not directly depend on path lengths and minimizing dilation costs is no longer a major concern [1]. However, since packets must compete for link bandwidth, the blocking time may become significant. Chittor and Enbody showed that link contention may degrade network performance substantially, especially in large networks [1, 2] or networks with heavy traffic. Now that path length is no longer a problem, link contention becomes a primary concern in ensuring efficient communication. A good measure that quantifies the level of contention in a wormhole-routed network, called ***path contention level*** (PCL), was introduced in [2, 4].

Random mapping is claimed to be good enough for second generation wormhole-routed networks [9]. In fact, small system sizes and high communication overheads are limiting the effect of contention from becoming serious in the current systems. However, as the network size increases and overheads are reduced, the contention problem will surface and seriously impact communication performance [2, 4]. It was shown that appropriate mappings that minimize contention will enable the network to support higher traffic with negligible degradation in performance [2, 4].

Nonetheless, there has not been enough emphasis on program mapping for wormhole-routed networks. As part of this thesis, we study one-to-one mappings among the most important topologies: lines, rings, hypercubes, square meshes and square tori. These topologies represent the communication structures of many applications in scientific computations as well as the topologies of many large-scale wormhole-routed networks [5]. Our mapping objective is to minimize the maximum PCL of a virtual-architecture program mapped to a given physical topology.

Path contention level of a path p connecting two communicating processes can be roughly defined as the number of other paths which share at least one link with path p. Path contention level represents the worst-case contention of a path. That is, it assumes that all competing paths are working at the same time. In reality, some of the competing paths may not route any messages at some time, depending on the application communication pattern. In any case, if we can minimize the path contention level of a path, the path will have less chance to collide with other paths. This effectively reduces blocking time of messages.

We make experimental comparisons between network performance of our mappings and that of random mappings. The experiments are carried out on the CPSS under different system and user-defined parameters. Simulation results show that our proposed mapping functions can significantly outperform random mappings in terms of communication performance, especially on large networks.

Chapter 3

System Architecture and High-Level Design

In this chapter, we first discuss our objectives for the design of the CPSS. We then present the modeling of multicomputer systems simulated by the CPSS and the programming model used in the CPPE to write application programs. We then describe the high-level design of the simulator, which is based on the adopted system model and programming model, and meets the proposed design objectives.

3.1 Design Objectives

The most important objectives considered in the design of the simulator are as follows.

1. Realistic modeling: The simulator should reflect accurately the behaviors of the multicomputer system as well as the performance of an application program on this system. So the design should follow realistic computation and communication models, and retain the essential characteristics of the multicomputer system.

2. Accurate simulation: The simulator should employ the functional simulation approach to simulate the execution of parallel application programs [13, 34, 25]. It should, at the same time, execute the application and simulate the behaviors of the underlying multicomputer system. The outcomes during and after each execution are thus the outputs of the application program, and information about its computation and communication performance.

3. Performance: Time of simulating the execution of application programs is also a crucial issue. We selectively left out low-level details so that the simulator can produce accurate outputs in a reasonable amount of time, especially for large applications. The granularity of instruction interpretation should be weighted carefully to meet both the accuracy and performance requirements.

4. Flexibility: The simulator should offer the user the freedom of changing system parameters for both computation and communication. This enables the user to tune the application to the underlying hardware at hand, or to obtain a thorough performance analysis of the application on various multicomputer systems having different characteristics. The flexibility of changing parameters should also come with convenience: the parallel program need not be re-compiled every time some parameter is modified.

   The design and implementation of the simulator should be modular and decoupled to easily accommodate future changes and enhancements.

5. Repeatability: Repeatability is necessary to study different phenomena in an execution at several levels of detail and from different perspectives. Repeatability is essential to provide a stable and reliable debugging environment that is not available on real multicomputer systems. Real parallel computers are nondeterministic in nature and, as such, rarely provides any form of repeatability; some bugs may not occur frequently enough for observation. Repeatability does not mean that the simulator can reproduce only one of the many possible executions of a nondeterministic application: the simulator should also be able to simulate multiple executions of an application when it is required to mimic the nondeterministic nature of real multicomputer systems and of parallel applications.

6. Correctness debugging: The simulator should provide end-users with convenient debugging tools and useful debugging information in order to test and debug application programs. After all, this is the main advantage for using a simulator rather than a real multicomputer which rarely provides any form of repeatability and supports only a very limited set of debugging tools. The debugging code embedded within the simulating code should not affect behaviors of application programs and their outputs.

7. Performance debugging: Computation and communication statistics of application execution should be provided to facilitate the study of parallel architectures, network characteristics, parallel algorithms and program mapping. Global statistics of an application (such as total execution time, speedup, total computation time and total communication time) allow the user to tune the application to a desired performance. Run-time data or traces (such as process creation/termination information, or message send/receive information) could be very useful in studying a particular aspect of the application, which cannot be captured by global statistics.

8. User-friendliness and portability: It should be easy to learn and use the simulator. An intended use of the simulator is to introduce the field of parallel computing to new learners. Also, the simulator should be portable to various platforms (such as workstations, PCs, or even on real parallel computers) with only very minor modifications to the simulating code.

3.2 Modeling the Multicomputer System

3.2.1 System Architecture

The simulator assumes a distributed-memory, wormhole-routed parallel computer. This parallel system consists of a number of processing nodes (processors) connected by an interconnection network. As depicted in Figure 2, each processor includes a processing element (PE), a local memory (LM) and a router (R). The processing element runs the application program which resides in the local memory together with the application data. The router is responsible for receiving incoming messages, injecting messages into the network, and forwarding them to destination processors.

This wormhole-routed multicomputer is characterized by the following parameters [14]: the communication delay L, the communication overhead o, the communication bandwidth g, and the number of processors P. Various aspects of the architecture are approximated by these four parameters. Parameters used by the simulator can be categorized to model L, o and g. If communication contention is absent, P is small, and messages are short, then L can be disregarded. Otherwise, L is decided by the level of contention, P, and message lengths. P also determines the obtainable speedup of the multicomputer system. Ideally, the speedup should increase linearly with the number of processors used to run an application. In reality, L and o prevents the linear increase of the speedup.

The cost model used by the simulator assumes that computation and communication are overlapped. In other words, the processing element and the router of a processor are working simultaneously during program execution instead of alternating their turns.

In our simulator, the characteristics of the multicomputer system are modeled by two sets of parameters. One set of parameters reflects the communication aspects of the system while the other set captures the computation aspects.

3.2.2 Communication Parameters

The time $T_{message}$ for a message to reach the destination is the sum of message startup overheads and communication latencies of its packets. $T_{message}$ is determined by

$$T_{message} = T_{mstartup} + p \cdot T_{packet}$$

where $T_{mstartup}$ is the startup cost of each message send, p is the number of packets contained in the message, and T_{packet} is the communication latency incurred by a packet.

The communication latency of a packet is in turn the sum of packet startup overheads and routing latency. Therefore,

$$T_{packet} = T_{pstartup} + T_{routing}$$

where $T_{pstartup}$ is startup overhead of a packet send and $T_{routing}$ is the routing latency of a packet.

In the absence of link contention, $T_{routing}$ can be approximated by

$$T_{routing} = (F/B)D + P/B$$

where F is the flit size (in bits), B is the channel bandwidth (in bits/second), P is the packet size (in bits), and D is distance between the source and destination nodes (number of hops).

If link contention is present, $T_{routing}$ is dependent on run-time conditions and is determined based on the routing latency of flits. The flit routing latency consists of the following components [26]:

- Buffer read time: the time needed to read the flit from the current buffer.

- Router decision time: the time required by the router to determine the next node on the path. This time depends on the routing algorithm and message address format. Router decision time applies only to the header flit: after the routing direction is determined, the remaining flits simply follow the header.

- Virtual channel allocation time: the time needed to allocate a free virtual channel of the next link where the flit will be deposited. This time is applicable only to the header flit, and depends on whether there are free virtual channels on the next link.

- Link scheduling time: the time required to schedule the physical link in order to know which virtual channel is allowed to use the link in the current clock cycle.

- Link delay time: the time taken by the flit to traverse the link and arrive at the next node.

- Buffer write time : thc time needed to write the incoming flit to the buffer at the next link.

All flits require buffer read time, link scheduling time, link delay time and buffer write time to move from one node to its adjacent node. In addition, header flits incur router decision time and virtual channel allocation time.

The following communication parameters are taken into account in the wormhole-routed network of the CPSS:

- Topology and size of the virtual and physical architectures. The simulator currently supports line, ring, hypercube, mesh and torus topologies. The system size can be any number provided that the host computer has enough memory to support all data structures of the CPSS. Most of these data structures are dynamically allocated and deallocated. We have been able to support systems with up to 4096 processors.

- Packet size. Large messages need to be split into smaller units called packets. Typical packet size ranges from 64 to 512 bits [23].

- Flit size. The flit length is often affected by the network size which determines the minimal length required to represent node addresses. For example, a 256-node network requires 8 bits per flit.

- Number of virtual channels per physical link. Several virtual channels can be multiplexed on a single physical link to time-share the link bandwidth.

- Buffer size. Each virtual channel of a physical link calls for a buffer to store the on-going flits of a packet. Many systems make use of one-flit buffers. Larger flit buffers can improve network performance and help to increase network size without increasing flit size [8].

- Message start-up overhead. This is the startup cost of each message send and due primarily to message buffer management.

- Packet startup overhead. This is the startup cost of each packet.

- Flit routing latency components as mentioned above. The components are buffer read time, link scheduling time, link delay time, buffer write time, router decision time, and virtual channel allocation time.

- Deadlock prevention. To prevent deadlock, we use bidirectional channels together with E-cube routing (for hypercube networks) or XY routing (for meshes). We also support deadlock-free deterministic routing for ring and torus; the algorithm was proposed by Dally and Seitz[15].

- Routing strategy. Our simulator has so far supported only non-adaptive routing (deterministic routing). The default routing schemes are E-cube routing for hypercube networks and XY routing for other topologies. The user can also specify the desired routing algorithm in the form of a routing table.

- Virtual channel allocation algorithm. When there are several outstanding messages waiting for a virtual channel to be allocated, our simulator uses first-in-first-out algorithm to decide which message will get the virtual channel. Among other algorithms are random allocation and priority-queue allocation which is usually used in real-time systems.

- Link scheduling algorithm. In the CPSS, round-robin scheduling is used for virtual channels of a physical link to time-share that link. Among other algorithms are random selection and priority-queue selection which is usually used in real-time systems.

Our selection of communication parameters is based on their importance and their influence on the accuracy of program execution simulation as well as the simulation time [Dally87, Chittor90, Chittor90b, Chittor91, Draper94, Kim94, Olk94]. For example, the simulator of a wormhole-routed multicomputer described in [Olk94] made use of some other detailed communication parameters. We have chosen to ignore those parameters because that simulator was designed to analyze communication properties of parallel application domains in order to find the most cost effective communication hardware configuration matching the requirements of the specific domain. Our simulator, on the other hand, is aimed at analyzing both computational and communication properties of the parallel program so that the user can obtain the best algorithm for the problem at hand or to tune the application to a specific underlying hardware. We have thus made a tradeoff between the feasibility of simulation time and the accuracy of the simulated hardware network.

3.2.3 Computation Parameters

Processor Clock Cycle

In the CPSS, the clock cycle is expressed in terms of relative timing. That is, a clock cycle is the minimal time unit. It is up to the user to define the duration of one time unit. All other computation time parameters are then specified using the minimal time unit. For example, an integer addition can be defined to take one time unit. If an floating-point operation is ten times slower than an integer addition then its cycle count is 10.

Instruction Groups

Intermediate instructions are classified into instruction groups to facilitate the assignment of cycle counts to the instructions. Each group has its own execution time. The major instruction groups are: integer operations, floating-point operations, I/O operations, function calls, process creation/termination, and read/write on channel variables (which is equivalent to message send/receive)

Context Switching

In the CPSS, many processes can run on the same processor to share the computation bandwidth of the processor using round-robin scheduling. Each process is given a time slice to utilize the processor just as with multitasking uniprocessors. Context switching among these processes would incur overheads on the target machine. Therefore, in the CPSS, context switching is also assigned a cycle count, and context switching time is a definable parameter.

Computation Quantum

The CPSS implements parallel execution of processes using time slicing: each application process is given a quantum to run until its quantum expires or it is put to sleep by some event. Application processes are scheduled in a round-robin fashion during every quantum . When every application process has finished its quantum, the global clock is advanced to the next quantum.

Computation quantum is determined based on flit routing latency that is the time a flit takes to move from one node to the adjacent node on the path. For example, if a flit takes 3 time units to advance to the next node on its path then the computation quantum is 3 time units. In this case, the CEM lets the processes run for 3 time units, and the network advances unblocked flits by one link. The processes will then run for another 3 time units, and unblocked flits will move forward by one more link, and so on. The computation quantum and the communication step are considered to be running in parallel. Computation quantum can be a fraction number. If the computation quantum is 0.5 time unit, the processes will run for one time unit every time the network advances unblocked flits by two hops.

Process States

Possible process states are:

- Ready. The process is ready to be scheduled for running.

- Running. The process is currently executing its code on a specific processor.

- Delayed. The process is put to sleep and the wakeup time is known. The process will be waken up by the scheduler when the wakeup time comes.
- Blocked. The process is put to sleep and wakeup time is not known in advance. The process will be unblocked by another process (e.g., one of its children) or an event (e.g., the arrival of a message in the wormhole-routed network).
- Terminated. The process has completed the execution of its own code.

3.3 Application Programming Model

Applications that are to be executed on the CPSS are initially written in the CPC language (Concordia Parallel C), an enhanced version of the C language used for parallel programming.

We support both *data parallelism* and *functional parallelism* programming. In data parallelism, the same computation is applied in parallel to different data items. This is in contrast to functional parallelism, in which several different computational activities are performed in parallel.

In particular, applications can be programmed using the SPMD (Single Program Multiple Data) model. In this model, every process of the application is running the same program. However, at a given time, each process may execute a different part of the program.

The application programs can be written using the virtual architecture approach. In this approach, the programmer writes the program in the topology most suitable for the problem in question. At run time, the simulator will map the compiled program onto a physical topology specified by the user, which can be different from the virtual topology. The CPC language also provides features that allow the programmer to specify a mapping from the virtual topology to the physical topology in the program.

Communications between processes are expressed in terms of channel variables [25]. A write to a channel variable represents a send, and a read represents a receive. Writes and reads from channel variables will be converted into sends and receives respectively by the CPCC (Concordia Parallel C Compiler).

3.4 CPSS High-Level Design

This section gives a high-level description of the simulator. The design is based on the multicomputer system model and the programming model mentioned earlier, and implemented in a way to meet the proposed objectives.

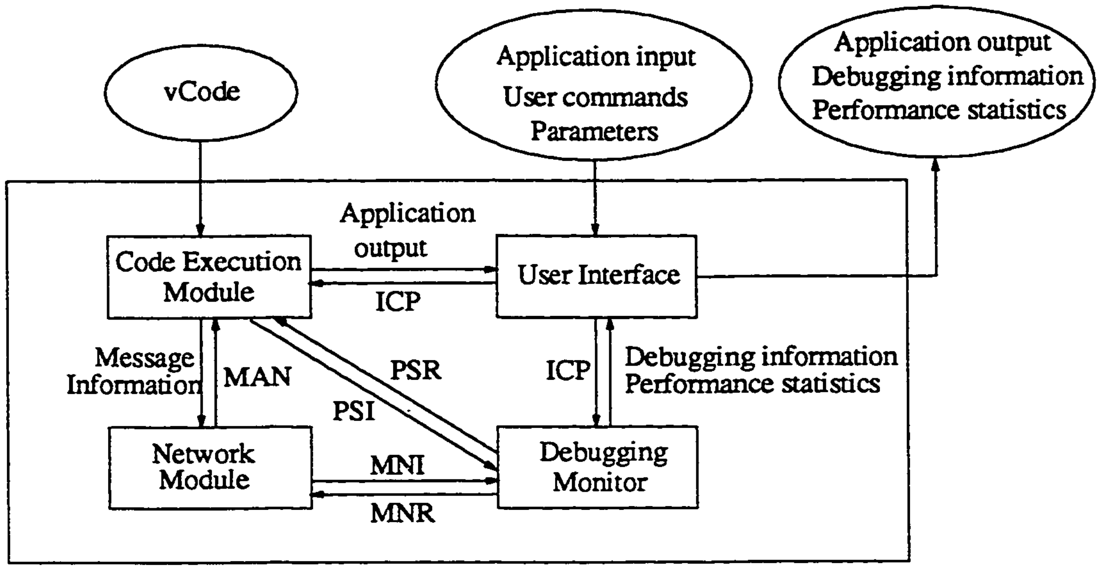
3.4.1 General Structure of the CPSS

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

The core of the CPCC is a compiler. After reading a parallel program written in the CPC language, the CPCC builds a complete abstract syntax tree to perform syntax and semantic analysis, and produces object code for a generic virtual machine. Such object code is called *vCode* in the CPPE. The vCode instruction set is defined based on an analysis of common operations of multicomputer systems. To produce vCode, the compilation process makes use of the virtual architecture and does not call for the physical architecture. The advantage of this design is that the CPC parallel program need not 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 (Figure 1).

The CPSS itself consists of two major components: the code execution module (CEM) and the network module. There are also two other utility modules interworking with the CEM and the network module. These utilities are the user interface and the debugging monitor. The interactions between the components of the CPSS are illustrated in Figure 3. The following subsections describe the roles and high-level design of the CPSS components and their interactions.



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

Figure 3: CPSS structure and operations

3.4.2 The Code Execution Module

The CEM plays the role of processing elements of a multicomputer system: it executes the parallel code specified by the parallel program. There is a global clock for the simulated multicomputer system which is updated periodically by the CEM. The CEM contains four main parts:

1. Mapper. The mapper maps the processors of the virtual architecture specified in the CPC program onto the processors of the physical architecture. We provide a library of optimal mappings whose objective is to minimize the maximum path contention level of the parallel program [2, 4]. Optimized and random mappings are also available. The mappings can be one-to-one or many-to-one.

2. Storage Manager. The job of the storage manager is to allocate simulated local memories to processes upon process creation and deallocate this space upon process termination. The storage manager also allocates/deallocates other kinds of dynamic memory blocks such as activation records upon function calls/returns, and buffers for messages of composite types (e.g. array, structure).

3. Process Scheduler. The process scheduler schedules processes for execution and updates their process structures according to changes in process and processor status, local clocks and the global clock. 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 process scheduler traverses the list of processes, and schedules one process at a time for execution. The execution starts with the instruction specified by the current value of the program counter of the process. The local clock of the process is updated after each instruction. The process runs until its quantum expires or it is put to sleep by some event. The process scheduler then schedules the next process for execution. When every application process has finished its quantum, the global clock is advanced to the next quantum.

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

The CEM contains four major data structures:

- List of parallel processes. Each parallel process created by the virtual-architecture program is associated with a structure PROCESS that contains all the information needed to run this process. Each process has a local clock that keeps track of the present time of this process. All PROCESS structures are placed on a linked list that is maintained and processed by the scheduler.

- Table of processors. This is an array where each element is a structure PROCESSOR. This array is dynamically allocated at the beginning of each run when the physical architecture is known. Each PROCESSOR structure records the status of and information related to a physical processor.

- Memory pool. This is a big array from which local memories of processors are allocated. The array accommodates local memories of all processors in use. The memory pool also provides space for activation records upon function calls, and buffer space for messages of type array or structure. Memory blocks (local memories of processes, activation records, message buffers) are allocated upon requests and returned to the common memory pool when they are no longer used. The first-fit allocation scheme [31] is used for management of the memory pool.

- Message buffers (channel buffers). This is an array where contents of messages are buffered, waiting to be read. A message of type array or structure is too big to be stored in this array. In this case, the actual contents of the message is stored in a block of the memory pool, and the pointer to this memory block is saved in the array of message buffers.

3.4.3 The Network Module

The network module is under control of the network manager. The role of the network manager is to

- allocate network resources to messages to be sent,
- route messages and deliver them to destination processors,
- detect and resolve deadlock, if any.

The main features of the wormhole-routed network manager are the reservation of channels and data paths by messages, the pipelined flow of flits, the release of the reserved resources by tail flits, and the release of flit buffers associated with each virtual channel. The network also uses the global clock mentioned above. In each quantum, all active packets that are not blocked are advanced by one link.

When a message needs to be sent, the sender process writes the message to a message buffer, invokes routine *WH_CEM_SENDS_MSG* to pass message information to the network manager, and continues with its execution (non-blocking send). The network manager will route the message using the information received from the sender. In our design, the network simulator does not route actual contents of messages. It simulates the movement of flits by advancing their ID numbers. When a message reaches the destination, the network manager notifies the CEM of the arrival of the message. The intended reader can then read the message from the message buffer.

The main data structures of the network module are:

- List of new messages. New messages which are being initialized for routing are queued at this 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 this list and appended to the list of active messages.

- List of active messages. This is a linked list of messages which are currently being routed through the network.
- List of active packets. This linked list contains packets belonging to active messages.
- Array of physical link structures. Each physical link structure stores information related to that link such as the link-request queue, number of occupied virtual channels, an array of LANE structures (a structure for each virtual channel of the link).
- Routing table. When a packet arrives at an intermediate node, the router uses the addresses of the current node and the destination to look up the routing table for the address of the next node on the path.

3.4.4 The User Interface

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 pass valid parameters or commands to the appropriate module (the CEM, the network module, or the debugging monitor). During execution of a parallel program, the user interface interacts with the debugging monitor to display performance statistics and debugging information. Program outputs are also transfered from the CEM to the user interface for displaying. A user manual is provided in Appendix B which describes the user interface of the CPSS in detail.

3.4.5 The Debugging Monitor

The debugging monitor is responsible for handling the debugging mechanisms. During execution of the parallel program, the CEM and the network manager regularly update the debugging variables. After each breakpoint and after the completion of program execution, the debugging monitor collects and processes the values of the debugging variables to generate performance statistics and other information about program execution.

3.5 Measures to Meet the CPSS Design Objectives

In this section, we discuss how the design and implementation of the CPSS meet the objectives stated in section 3.1.

3.5.1 Accurate Simulation

The CPSS uses the functional simulation approach to simulate execution of parallel programs on a multicomputer system [13, 34, 25]. The level of detail of code interpretation decides the accuracy and the performance of the simulation. A fine granularity of interpretation offers an accurate simulation at the cost of low simulation speed. The tradeoff between accuracy and simulation time has been considered carefully in the design of the CPSS to provide accurate simulation results within an acceptable amount of running time. The degree of accuracy is adjustable and decided by the definition of vCode instructions, their associated costs, and the level of detail of interpreting each instruction.

Accurate simulation of parallel architectures is also promoted by configurability of most computation and communication parameters. Configurable parameters allows the user to accurately simulate a particular multicomputer system. The user only needs to set the values of system parameters to those belonging to the architecture to be simulated.

3.5.2 Performance

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

We analyzed basic operations of existing parallel architectures and constructed a set of parallel primitives which are common to most target multicomputer systems. Parallel primitives are simulated at the functional level with a reasonable abstraction to tradeoff between simulation accuracy and simulation time. Because the level of interpretation is higher than machine instruction level, the simulation is much faster compared with the traditional functional simulation.

The entire simulation system, including the application program, is run by a single process. The simulation does not incur any context switching on the host, and thus saves simulation time. In Proteus, which uses light-weight processes to simulate application processes, a context switch takes 3 microseconds [12]. Tango uses UNIX processes, so context switching time is as high as 250 microseconds [19].

The simulated wormhole-routed network does not route actual messages. The routing is simulated using only message information. Moreover, the network uses an approximate version of round-robin scheduling to speed up the simulation time.

3.5.3 Flexibility

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

Virtual-to-physical architecture mapping at run time

The application program is written using the virtual architecture which should be the most natural and efficient architecture for the application. At run time, the virtual architecture will be mapped to a physical architecture. If the user does not specify any physical architecture, the default is the virtual architecture itself. In this case, no mapping is needed.

If the user specifies a physical architecture, a mapping is needed to assign the virtual nodes to the physical processors. The user can select a mapping function from the mapping library supported by the CPSS. The mapping library includes

- Optimal mappings which offer the best communication performance. A set of optimal mapping functions for wormhole-routed networks are provided in Chapter 6.

- Approximative mappings which are optimized to provide good communication performance.

- Random mappings

The flexibility is extended to allow the user to use his/her own mapping. The mapping is generated by the user, stored in a file which the simulator will read in to map the virtual nodes onto the physical processors. Note that the mappings can be both one-to-one and many-to-one.

All the above changes (physical architecture and mapping) require no modifications to the simulator or the application, and thus no re-compilation. This is an ability unique to the CPSS among existing multicomputer simulators. In the case of Proteus simulator, to change the topology of the physical architecture, the network module must be modified to the new architecture [12]. The whole simulator is then re-compiled and re-linked. The same procedure is applied to the EPPP simulator if the topology of the physical architecture needs to be changed [35].

With Multi-Pascal and its simulator, there is no support for virtual architecture [34, 25]. The architecture defined in the program is also the physical architecture. Processes are mapped onto processors in the application program. If the physical architecture or the mapping need to be changed, the application program must be modified and re-compiled.

Simulating a wide range of multicomputer systems

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

The vCode instruction set of the CPSS is constructed using common operations of multicomputer systems. The cost of each instruction can be adjusted to simulate a specific target. Instruction functionality can be modified and new instructions be added easily due to the modular and decoupled implementation of the simulator.

The CPSS simulates a wide range of topologies: line, ring, mesh, torus and hypercube. System size can be extended up to thousands of nodes and is limited only by the memory capacity of the host computer. The simulator also permits users to change most system parameters to define a specific architecture.

Large set of configurable parameters

Almost all computation and communication parameters can be configured by the user. The computation parameters listed in section 3.2.3 are all user-modifiable. Besides the physical architecture and the mapping, the following communication parameters are configurable: packet size, flit size, number of virtual channels per physical link, virtual channel buffer size, startup overheads, flit routing latency, header flit overheads, and routing algorithm.

The user can change values of parameters within the same simulation session as often as needed. No re-compilation is needed: the same vCode of the application is always executed.

The large set of configurable parameters enables the CPSS to simulate a wide range of wormhole-routed networks and multicomputer systems. Tuning system parameters allows the analysis of an application or an architecture to be more thorough and accurate.

Modularity and expandability

The design and implementation of the simulator are modular and decoupled. Future changes and enhancements to the simulator would be quick and easy. For example, a new vCode instruction can be added easily to the simulator. We only need to write the simulating routine for the new instruction. No other modifications are necessary (unless the new instruction interacts with existing instructions, in which case other changes are required to capture the interactions).

The network module is decoupled from the CEM. The interface between these two modules are clean and well-defined. It is possible to develop a simulator of another type of network (such as packet switching) independently. It would then be easy to integrate the new network module into the CEM to simulate packet-switched multicomputers.

3.5.4 Repeatability

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

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

In the CPSS, multiple executions are implemented by varying the relative processor speed. Race conditions can thus 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.

A pseudo random number generator is used to determine relative processor speeds. Using the same seed for all executions provides the determinism needed for repeatability. On the other hand, if we vary the seed values, relative processor speeds change accordingly. This results in different executions of the same application.

Proteus simulator [12] applies the same concept of using a pseudo random number generator to implement repeatability and multiple executions. However, the seed is used to randomly choose between two requests with the same timestamp. Different choices generates multiple executions.

The implementation of the CPSS is different from that of Proteus. If we imposed a choice on any two requests of the same kind with the same timestamp, this would slow down the simulation time since the race condition was being simulated at a very low level (i.e. at the level of requests). Instead, we make use of an alternative which is to vary relative processor speeds. This meets the same objective (i.e. repeatability and multiple executions) while offering a higher performance.

Repeatability is critical to provide a stable and reliable debugging environment. It also helps to study an application at various levels of details and from different angles. Multiple executions are essential to study nondeterministic applications, applications which demonstrate different behaviors depending on results of race conditions. An example of such applications is concurrent branch-and-bound search algorithms. For example, a concurrent search algorithm would require multiple executions so that the user can gather a distribution of execution times, which allows for a much more accurate view of the effectiveness of the algorithm. Multiple executions are also very useful to test the robustness of deterministic applications. A deterministic program should work correctly under different outcomes of race conditions occurring in the system.

3.5.5 Correctness Debugging

Another advantage of the functional simulation over the other simulation technique is easy debugging. Since parallel object code instructions are interpreted at the functional level, it is convenient to insert debugging code inside the interpretation code. Unlike the case of direct simulation technique, the amount of inserted debugging code does not affect simulation results at all.

The CPSS supports the following debugging tools:

- Set and clear instruction breakpoints.
- Set and clear time breakpoints.
- Step instruction by instruction; step through two or more instructions.
- Set and clear trace variables.
- View the trace variables.
- 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).
- View the vCode corresponding to selected lines 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 function that is currently executed by this process
  - the line in the source code that is currently executed by this process.

The simulator also provides information about network deadlock or deadlock incurred by program logic (for example, all processes are waiting to receive messages but there are no instructions in the program that send messages), if any. For network deadlocks, displayed information includes the physical links and the messages involved in the deadlock, and their status. In the case of program-logic deadlocks, the processes involved in the deadlock and their status are displayed.

3.5.6 Performance Debugging

The performance statistics produced by the CPSS are:

- parallel execution time of the program
- sequential execution time of the program
- execution time of any portion of the program
- computation time of the program (i.e. time the program spent on computation tasks)
- communication time of the program (i.e. overheads incurred by message sends and receives)
- processor utilization
- profile of processor utilization as a function of time

In addition, time-dependent data and traces are available for in-depth analyses. Examples of time-dependent traces are:

- process creation/termination information (time, processor number, parent process ID)
- message sends/receives (time, source node, destination node, message length)
- message routing (path, time traces)

These data records are logged into files and can be disabled or enabled as the user wishes. Time-dependent data and traces are provided at different levels of detail as requested by the user. It is also easy for users to add their own traces to the simulator code in order to capture other time-dependent data as they want.

3.5.7 User-Friendliness and Portability

To make the simulator user-friendly, we reuse the debugging concepts of sequential programming environments. For example, a parallel application is first compiled by the CPCC and then executed by the CPSS. Debugging tools are similar to those of sequential programming environments such as breakpoints, trace variables, file listing, etc.

To support portability, the simulator is written entirely in C. It is intended to be running on any host machine which has a C compiler. Compilation conditional flags are used to adapt the simulator to different versions of C compiler. Currently, the simulator can work on UNIX workstations and PCs.

Chapter 4

The Code Execution Module

This chapter presents the design and implementation of the code execution module (CEM). The discussion focuses on the simulation of the most important entities of a multicomputer system, which are: processing elements (processors), local memories of the processors, processes, and communication channels among processes.

4.1 Processors

4.1.1 Virtual Processors versus Physical Processors

In the CPPE (Concordia Parallel Programming Environment), we distinguish two kinds of processors: *virtual processors* and *physical processors.* The user writes an application using the architecture most natural and efficient to program performance. This architecture is referred to as virtual architecture, and its processors are called virtual processors. For example, the natural topology for matrix multiplication should be 2D-mesh or torus.

The topology and size of the physical machine may not match that of the virtual architecture. Processors constituting the physical system are physical processors. At run time, the virtual processors are mapped to the available physical processors. For instance the matrix multiplication program may be mapped to run on a hypercube multicomputer.

The mapping objectives are to minimize communication cost among communicating processes, and to balance the workload among physical processors.

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

1. Process-to-virtual-architecture mapping. The mapping can be one-to-one and many-to-one. Often in the application program the user specifies the ID of the virtual processor on which a process will run. The virtual processor will be mapped to a physical processor at run time.

   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 since a virtual processor is not needed in this case. The physical processor allocated to the new process is determined by a *default processor allocation algorithm.* The default allocation criteria is to balance the work load among existing physical processors.

2. Virtual-to-physical-architecture mapping. At run time, the user can specify the desired physical architecture for running the compiled virtual-architecture program. The user is asked to select a mapping function provided by the CPSS

mapping library. Random mapping is also available. The user is also allowed to import his/her mapping from a file and store it in the *mapping table.*

If the user does not specify a physical architecture, the default physical architecture is the same as the virtual architecture.

In the CPPE, many processes are allowed to time-share the computation bandwidth of a processor. Processes residing on the same processor are scheduled in a round-robin fashion to run their code. Context switching among these processes incurs a configurable cost.

4.1.2 Processor Numbering

Processors (virtual or physical) are identified using absolute IDs. Absolute IDs are computed as follows.

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

Simulation of program execution utilizes absolute IDs so that simulation routines are generic and can be used for all types of topology. Only the mapping functions need to use Cartesian IDs (for meshes and tori) or binary IDs (for hypercubes).

4.1.3 Data Structures

At run time, virtual processors are mapped to physical processors and the mapping is recorded in the mapping table. Only physical processors need be simulated. Information required to simulate a physical processor is stored in a structure called *physical* *processor descriptor* (PPD). Every PPD contains the following fields:

- *status*: processor status that can be one of the following:
  - NeverUsed: the processor has never been used since the program execution started.
  - Empty: the processor has been used, but currently there are no processes working on it.
  - Occupied: there is at least one process currently working on the processor.
  - Reserved: a newly created process has reserved a place on this processor (the processor may currently run other processes). When the new child process starts running, the processor status will be changed to Occupied.

  Processor status is used only for the purpose of allocating default processors to newly created processes.

- *nbrProcesses*: number of processes currently sharing the processor.
- *runProcess*: pointer to the PCB of the process currently using the processor.
- *startTime*: time at which the currently running process is scheduled to use the processor. This field is used to schedule processes sharing the same processor.
- *virTime*: the running time accumulator of the processor. This field is used to schedule processes sharing the same processor, and for statistical purposes (e.g. calculating processor utilization).
- *speed*: processor speed, that is used to study the undeterministic nature of parallel program execution (section 3.5.4).

When the program starts execution, an array of PPDs is allocated, one array entry for each physical processor. The array size is the size of the physical architecture. The array is indexed by absolute IDs of physical processors. This array (`physProsorTable[]`) is referred to as the *table of physical processor descriptors*.

The data structure of physical processor descriptors in C is as follows.

```
typedef struct
{
 ProcessorState status;  /*processor status*/
 int nbrProcesses; /*number of processes running on the processor*/
 ProcDesPtr runProcess; /*pointer to PCB of current running process*/
 float startTime; /*starting time of currently Running process*/
 float virTime; /*running time accumulator*/
 float speed; /*speed multiplication factor; to vary processor speed*/
} PhysProc_Entry;
```

The mapping from virtual processors to physical processors are computed and stored in a mapping table (`virPhyMapTab[]`).

4.1.4 Physical Processor Allocation

In the application program, the user can map a new process to a virtual processor by specifying the virtual processor ID. In the following example, three processes are created and mapped to virtual processors with absolute IDs 1, 2 and 3 respectively.

```
for (i = 0; i < 3; i++)
   fork (i+1;) Compute(i);
```

Before the parent process creates a *fork* child process, it evaluates the expression representing the ID of the virtual processor on which the child will run. In the above example, the parent process calculates the value of $i + 1$. This value is the absolute ID of the child's virtual processor. The virtual processor ID is then converted to the physical processor ID using the mapping table. The new child will be spawned on the resulting physical processor.

If the user does not specify the virtual processor ID for a new child, the parent process skips the process-to-virtual-processor mapping. The parent will determine a default physical processor for the child by executing vCode instruction *DefaultProc*. Assuming that the program has n processors numbered from 0 to $n-1$ using absolute IDs, the algorithm of instruction *DefaultProc* is as follows. The parent traverses the table of physical processor descriptors. The first *Empty* processor found will be assigned to the child. If there is no *Empty* processor, the first *NeverUsed* processor found is chosen. If there is no *NeverUsed* processor either, among the *Occupied* and *Reserved* processors, the one with the least number of processes is selected. The status of the selected processor is then set to *Reserved*, and the child will be assigned to this physical processor.

The objective of *DefaultProc* algorithm is to balance the load among physical processors. To optimize communication performance of the program, the user should explicitly map new processes to virtual processors in the CPC program.

4.1.5 Context Switching

Processes sharing the same physical processor are scheduled to use the processor in a round-robin fashion. The processes are given equal time slices (called *switchLimit*) to run. When the *switchLimit* of a process p expires, another process will take over the processor.

However, if p is currently using the processor and has higher priority than the other processes, then p is allowed to continue to run even when its *switchLimit* has expired. The use of process priority would facilitate the simulation of real-time systems in the future.

When the running process completes execution of an instruction, the running time accumulator (field *virTime* of the PPD) of the processor is incremented by the cycle counts of the instruction. When a process is scheduled to take over the processor, field *startTime* of the PPD is set to the current value of *virTime*. The currently running process has used up its allocated computation bandwidth if $virTime \geq startTime + switchLimit$.

4.2 Memory Management

4.2.1 Physical View of Local Memories

In this section, we discuss how memory management is performed on real multicomputers. Local memories will be simulated based on this physical memory model.

In our model, several application processes are allowed to time-share a physical processor. Thus the corresponding local memory is shared by many processes. The local memory contains only program variables and other run-time data (e.g. activation records, dynamically allocated data structures).

When a new process is created on a processor, it is allocated a memory block from the local memory, 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 deallocated on the function return.

The size of the given memory block depends on run-time conditions and space requirements of the process execution. We assume that the allocated memory block is large enough for the process to run until normal termination.

Since several processes may be running on the same processor, memory protection must be available. The allocated memory block of each process is delimited by two registers: *base* and *limit* registers which contain the starting and ending physical addresses of the block, respectively. Figure 4a illustrates the local memory of a processor on which three processes are running. Each process owns a memory block and a corresponding pair of *base* and *limit* registers.

We now consider how a process makes use of its memory block during execution. When the process starts execution, it uses the memory block as a big working stack for expression evaluations or for temporary run-time data. A register (*top*) is needed to record the current top of the working stack (Figure 4b). If the process calls a function, an activation record is allocated and it will sit on top of the current working stack. The value of register *top* is saved in the activation record because the working stack

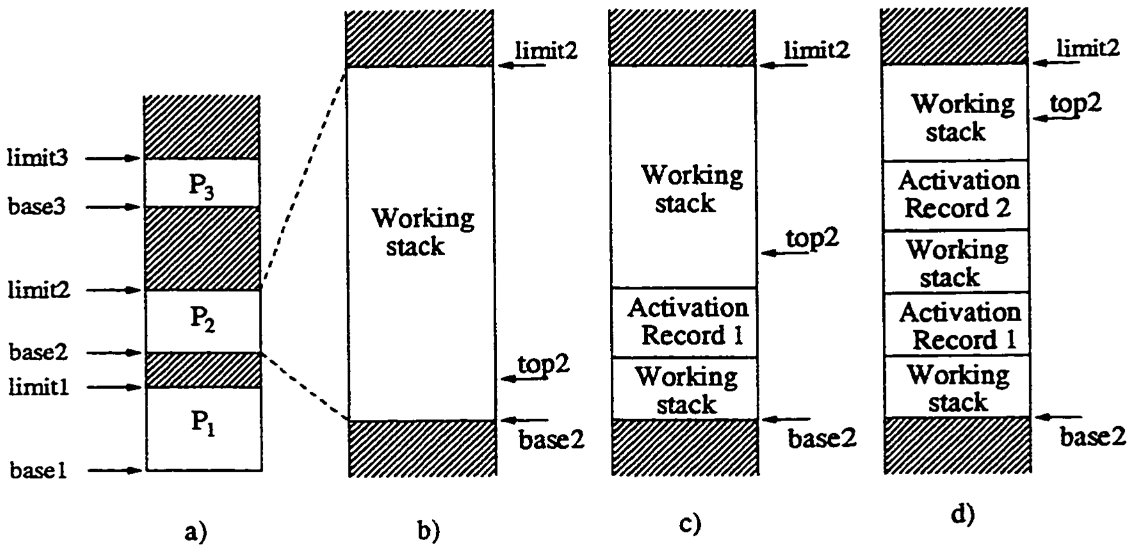


Figure 4: Physical view of local memories

now grows above the activation record (Figure 4c). If inside this function another function is called, then a second activation record is allocated. The current value of register *top* is saved in the second activation record, and the working stack moves above the second activation record (Figure 4d). When a function returns, the previous value of register *top* is restored, and the activation record is disposed. The working stack then returns to the previous position using the restored value of register *top*.

4.2.2 Design Choices

The first design choice that would come to mind is to implement local memories exactly as they are on a real multicomputer. That is, each simulated processor has a fixed-size local memory. The advantage of this design is that it reflects the exact image of local memories on real multicomputer systems, and the implementation is very simple.

The issue is how large a simulated local memory should be. If the memory size is small, some processors with considerable load may run out of space quickly. If the size is large, a lot of space may be wasted, and we may not be able to simulate very large systems. Also, if work loads on processors are not balanced, it may happen that some processors have idle memory space while others are running out of memory.

Therefore, we selected the dynamic allocation scheme. That is, memory blocks are allocated only when required and on a per-process basis. Consequently, if a processor is unused, the size of its local memory is logically 0. When a process is created on this processor, the process is allocated a memory block for execution.

The next issue is how much space should be allocated to a process. We do not know in advance how much space a process would need. Allocating fixed-size blocks would not be a good choice because some processes may waste unused space while others do not have enough memory to run. Therefore, we also allocate memory space to processes upon requests. More specifically, when a process is created, it is given space for a working stack so that the process can evaluate expressions and maintain temporary data. When the process calls a function, an activation record is allocated. During execution of the process, dynamically allocated data structures are also granted on a request basis.

When a function call finishes, the corresponding activation record is returned to the system for reuse. Similarly, dynamic data structures are returned when the process releases them. The system also reclaims the working stack when the process terminates.

4.2.3 Implementation

We make the following decisions in the implementation of local memories:

- Implementation of local memories does not include storage for program code. Program code is stored in a separate array and is shared by all processes. This implementation of code storage is economical since processes running the same code fragments can share one copy of the code fragments.

- Implementation of local memories does not include storage for process control blocks (PCBs). PCBs are stored in a separate array for use by the simulator.

- Stacks grow from the lowest address to the highest address.

Based on the selected design, memory blocks are allocated to processes upon requests. There are three kinds of memory requests: request on process creation, request on function call, and request for dynamically allocated composite-type data structures (i.e. arrays and structures).

```
typedef struct {  /*basic data entry*/
   char type;    /* tag 0: int; 1: float */
   union {
        int intValue;
        float floatValue;
   } val;
} basicValue, *basicValuePtr;

basicValue storageValue[STORAGE_SIZE];  /* memory pool */

int storageOwner[STORAGE_SIZE]; /*processor owners of memory words*/
```

Figure 5: Data structures of the memory pool in C

Memory blocks are taken from a common *memory pool* and distributed to requesting processes. The memory pool is a fixed-size array. Each entry of the array is considered to be a word. An integer or a character requires a word. A pointer is treated as an integer, thus needs one word as well. A word can also store a float number. To implement this, every word is associated with a tag indicating whether to interpret the word as an integer or a float. The data structure in C of the memory pool is shown in Figure 5. The memory pool is array `storageValue[]`.

Since the memory pool distributes memory blocks to all processes running on different processors, there must be a mechanism to tell to which physical processor a word belongs. An array parallel to array `storageValue[]` is used to store locations of memory words. This array is called `storageOwner` and of the same size as array `storageValue[]` (Figure 5). Entry `storageOwner[i]` contains the ID of the physical processor who owns the memory word `storageValue[i]`. Figure 6 shows the structure of two arrays `storageValue[]` and `storageOwner[]`.

The location of a variable (i.e., the physical processor on which the variable resides) is needed to determine

- the validity of an access: a process is not allowed to access a variable on a remote processor;

- the validity of a channel read: only the channel owner can read from that channel;
- the destination of a message send from a channel write.

When memory blocks are no longer needed (e.g. due to process termination, function return), they are returned to the memory for reuse. Thus fragmentation may exist: free and in-use blocks are interleaved in the memory pool. In the CPSS, memory blocks are allocated using *first-fit* allocation. Compared with the other schemes like *best-fit* and *worst-fit*, first-fit performs as well as best-fit and better than worst-fit in terms of storage utilization. Moreover, first-fit is generally faster than best-fit and worst-fit [31].

The simulator keeps track of free memory blocks for reuse. Each free memory block is delimited using two variables: *start* which contains the starting address of the block (memory pool absolute address), and *size* which is the block size (Figure 6). Free blocks are managed using an ordered linked list of structures, each structure recording the starting address and size of a free block (Figure 6). The linked list of free block structures is ordered in the increasing order of starting addresses (field *start*). When using first-fit allocation scheme, this order tends to reduce fragmentation because any two blocks allocated consecutively have more chance to be adjacent.

4.2.4 Memory Management for Processes

Allocation On Process Creation

When a parent process creates a new child, the parent requests a working stack for the child. The child will use this stack to store run-time data (such as the *forall information block* which will be described in section 4.5.2), or to evaluate expressions. On a real processor, the size of the working stack is unlimited (or limited by the upper bound of the physical memory block given to the process as illustrated in Figure 4b). In the simulator, the allocated working stack is part of the memory pool which is shared by all processes. Thus a working stack needs a limit. On the other hand if it is too small, a process may exceed the stack limit during execution, and cause a run-time error. The CPSS allows users to adjust the working stack size, which is a user-definable parameter. The very first working stack granted to a new child is called

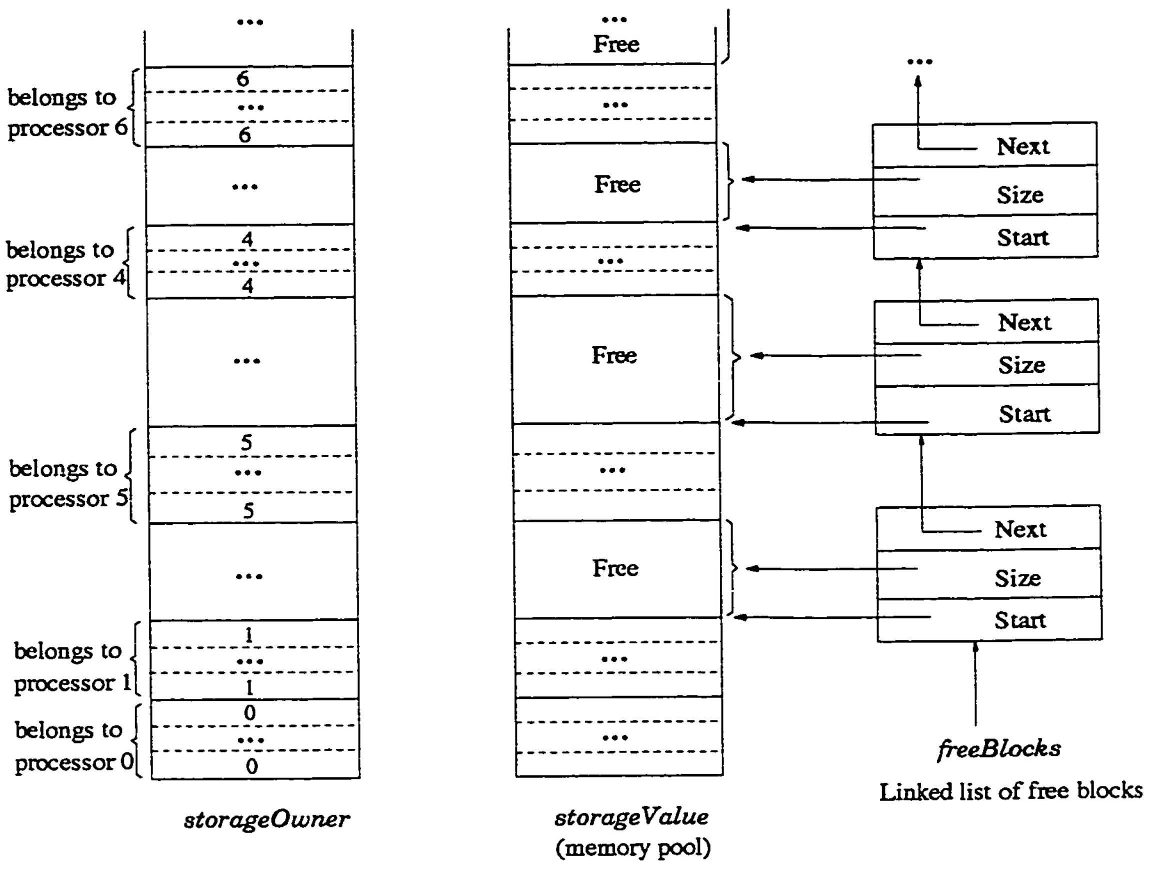


Figure 6: Implementation of the common memory pool

a *process frame*. Memory space allocated to a newly created process p is depicted in Figure 7a.

Using the CPC language, the code to be executed by a new child is denoted by a CPC expression. The expression can be a single statement, a function call or a block statement; examples are given in Figure 8. If process p's code is only a single statement involving no function call (Example 1, Figure 8), p does not require any more space. It simply executes the statement, terminates and returns the process frame to the system.

Allocation On Function Calls

If process p's code contains a function call as in Examples 2 and 3 in Figure 8, p will request space for the activation record and another working stack (Figure 7b). The second allocated working stack is used for expression evaluation or for temporary run-time data while the process is inside the called function. The activation record and its accompanying working stack are always adjacent, and they form a *function frame*. Note that this function frame of process p may not be adjacent to p's process frame. This is because the memory pool is shared by all processes, and the pool management is based on a linked list implementation.

If process p calls a second function while it is inside the first function, p will be given a second function frame, as illustrated in Figure 7c. p now owns two function frames and one process frame.

A new child's code can be a single function call as in Example 4 in Figure 8. In this example, right after being created, the child process calls function *RankSort*(). The call results in a function frame to be allocated to the new child, just as with other function calls. Memory space belonging to the child is as in Figure 7b.

The code of a new process can also be a block statement as in Example 5 in Figure 8. The CPSS treats a block statement as a function with no parameters and no name. This pseudo function may have local variables though. In the given example, the new process calls a pseudo function that has two local variables. Similar to the case of real function calls, the new child is granted a function frame (Figure 7b).

In summary, every time a function is called, a function frame consisting of an activation record and a working stack is allocated to the calling process.

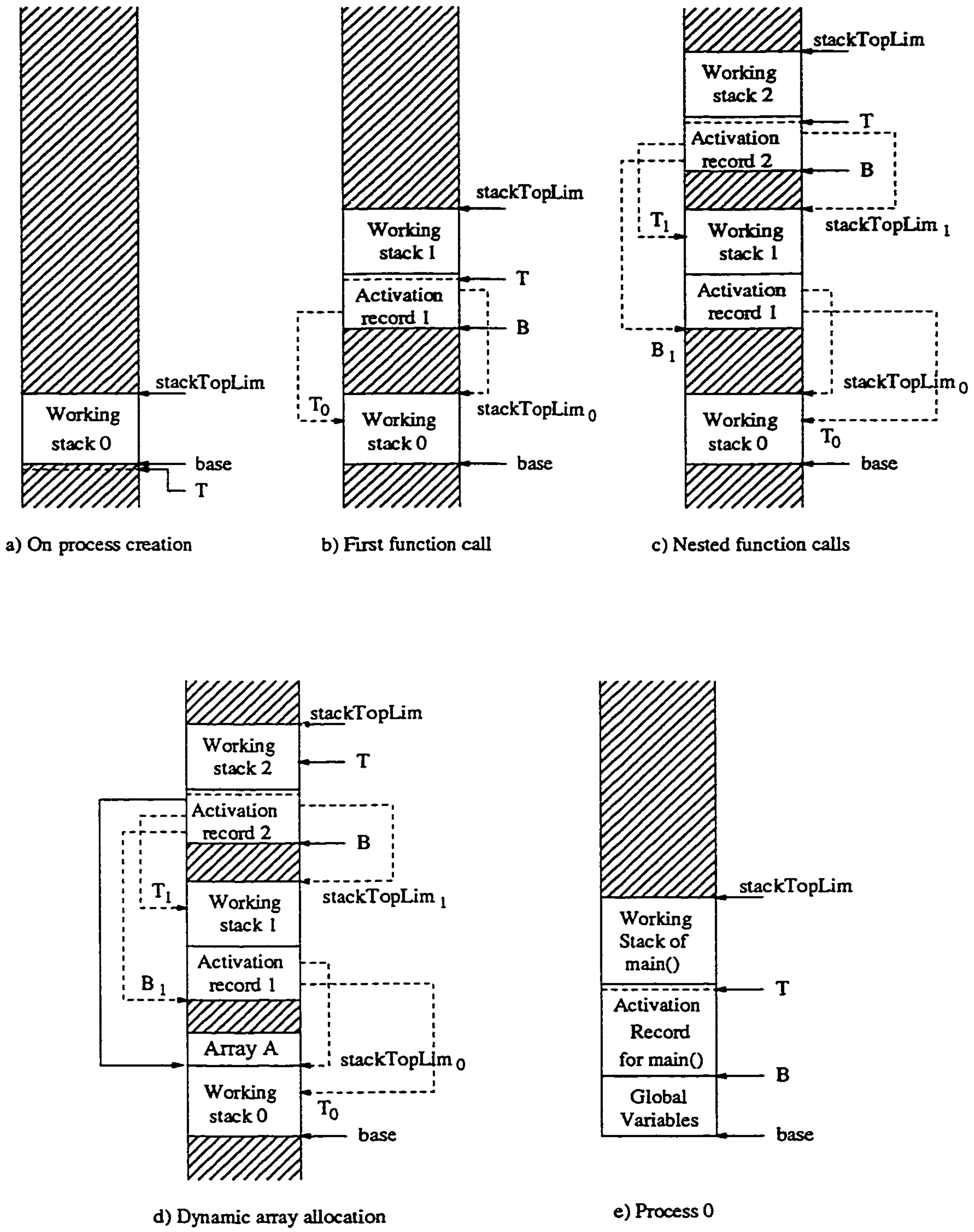


Figure 7: Memory management for processes

*/*Example 1: New child's code is a single statement without function call*/*

```
fork if (1)
   printf("Hello world!");
```

*/*Example 2: New child's code is a single statement with function call*/*

```
fork if (ReturnTrue())
   printf("Hello world again!");
```

*/*Example 3: New child's code is a single statement with function call*/*

```
fork while(1) {
   int A[10];
    ...
   RankSort(A);
    ...
   }
```

*/*Example 4: New child's code is a function call*/*

```
fork RankSort(A);
```

*/*Example 5: New child's code is block statement*/*

```
fork {
   int i, j;
   scanf("%d", &i);
    ...
   }
```

Figure 8: Examples of a new child's code

Dynamic Data Structure Allocation

During its execution, process *p* may request a dynamic data structure. For instance, *p* executes a *malloc*() to create an array *A*. A memory block of the size of array *A* is allocated to *p*. The block is granted using first-fit allocation scheme, as with other allocated memory blocks. The starting address of the array is recorded in variable *A*. This example is depicted in Figure 7d.

Deallocation

When a function returns, the corresponding function frame is returned to the memory pool. The same deallocation is performed when a composite-type data structure is freed. Similarly, when a process terminates, the process frame is released for reuse.

Process 0

Process 0 is somehow different from other processes. It is the first process created when program execution starts, and it has no parent. Global variables belong to process 0. Therefore, when the program starts running, the memory pool allocates the very first memory block (starting at address 1) to process 0 for global variables (Figure 7e). All process 0 does is to call function *main*(). Thus a function stack frame is allocated next for the call to *main*() (Figure 7e). Inside function *main*(), process 0 may create the first generation of child processes which will be granted memory blocks as described earlier.

Process Memory Management

As a process is running, its own memory space is maintained and handled by four pointers which are stored in the process control block (PCB):

- *base*: pointer to the starting address of the process frame allocated when the process is created. The value of this pointer never changes for the lifetime of the process.

- *stackTopLim*: pointer to the upper bound (the highest address) of the current working stack. When a function is called, the current value of *stackTopLim* is saved in the activation record of the function call. *stackTopLim* is then set to point to the upper bound of the new function frame. When the function returns, the previous value of *stackTopLim* is retrieved.

- T: pointer to the top of the current working stack. T is incremented when a value is pushed onto the stack and decremented when the top entry is popped off. Every time the process calls a function, the value of T is saved in the activation record of the function call. The pointer is then used to manage the new working stack. When the function exits, the previous value of the pointer is restored.

- B: pointer to the starting address of the current function frame. If another function f is called inside the current function, the value of B is saved in the new activation record allocated to f. B is then updated to point to the starting address of the new function frame of f. When function f returns, the process restores the previous value of B.

Figure 7 illustrates the use of these pointers. When a new process is created, it is allocated a process frame. *base* and *stackTopLim* point to the starting address and the highest address of the process frame, respectively (Figure 7a). T is initialized to $base - 1$.

Every time a process calls a function, the previous values of B, T and *stackTopLim* are saved in the newly allocated activation record. In Figure 7b and c, the dotted arrows denote the saved values of these registers. Pointers B and *stackTopLim* are then set to point to the starting address and the highest address of the function frame, respectively. T is set to the current stack top, which is the highest address of the activation record (Figure 7b and c).

When program execution starts, the pointers of process 0 are set as follows. *base* is set to address 1, which is the starting address of the memory block reserved for global variables. Since a function frame is also allocated immediately for function call to *main*(), B and *stackTopLim* are initialized to point to the starting address and the highest address of the function frame, respectively. T is set to the current stack top, which is the highest address of the activation record (Figure 7e).

4.2.5 Memory Management for Function Calls

Activation Record Description

When a process p calls a function, a function frame consisting of an activation record and a working stack is allocated to p. The activation record provides space for function parameters, local variables inside the function, and call/return control information. The structure of an activation record is depicted in Figure 9.

The call/return control information block is 9-word long and contains the following fields listed in the order from the lowest address to the highest address (Figure 9).

1. Function return value.
2. Current value of the caller's program counter.
3. Static link: pointer B of the lexical parent of the called function.
4. Dynamic link: the caller's B. B will then be set to point to the lowest address of the new function frame. The currently active function frames of process p are linked together by the pointers stored in this field (Figure 7b and c). The linked list represents the current *dynamic link chain* of p's execution.
5. Index of the called function in the identifier table.
6. Reference counter of this activation record. This field is only used when simulating shared-memory multiprocessors. We list it here to provide a complete picture of the activation record.
7. Size of the new function frame.
8. Current value of the caller's T.
9. Current value of the caller's $stackTopLim$.

Function Calls

The code sequence generated for a function call is as in Figure 10.

vCode instruction $NewFrame$ first allocates a function frame. It then saves the current value of pointer T into the activation record just allocated. Pointer

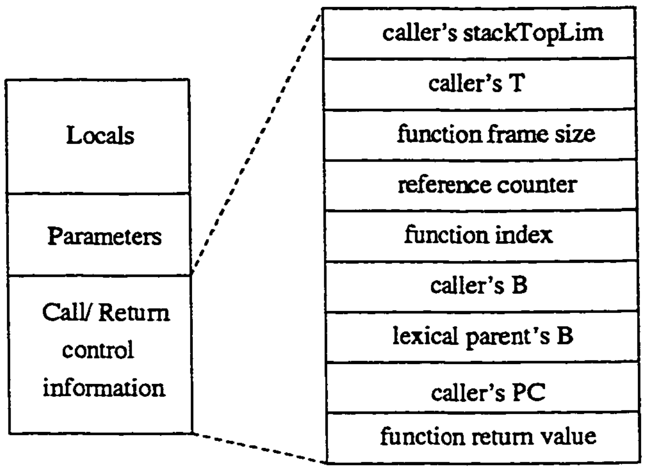


Figure 9: Structure of an activation record

NewFrame; */*allocate a new stack frame for the function*/*
Evaluate the parameters;
Call; */*call the function*/*

Figure 10: Function call (vCode instructions are in bold face)

stackTopLim is updated to point to the highest word of the new function stack frame. T is also updated to work with the new stack: T points to the highest address of the activation record.

The process continues by evaluating the parameters. Their final values are pushed onto the new stack, at their appropriate locations in the activation record. The process then executes instruction *Call*. This instruction saves call/return control information, updates pointer B to point to the lowest address of this function frame, and sets the program counter to the starting code address of the function body.

When the function exits, the saved values in the activation record are restored to allow the process to go back to the caller's context. The function return value is pushed on top of the caller's stack, and the callee's function frame is freed.

4.3 Processes

In this section we discuss how parallel processes and their execution are simulated. We also describe data structures needed to manage and run simulated parallel processes.

4.3.1 Data Structures

Process Control Block

Every application process is associated with a process control block (PCB) that stores various information needed for its execution. A PCB is dynamically allocated upon the creation of a new process, and deallocated when the process terminates. The PCB of a process p contains the following fields (field descriptions include reference to the sections that describe the fields in more details):

- *processID*: process ID
- *PC*: program counter
- *state*: process state, which can be one of the following states: *Ready*, *Running*, *Delayed*, *Blocked* and *Terminated* (section 4.3.3).
- *priority*: process priority (sections 4.1.5 and 4.3.4). Process priorities are used to schedule processes running on the same processor.
- *parent*: pointer to the PCB of the parent process
- *base*: pointer to the lowest address of the process frame (section 4.2.4)
- *T*: pointer to the current top of the working stack (section 4.2.4)
- *B*: pointer to the lowest address of the current function frame (section 4.2.4)
- *stackTopLim*: pointer to the highest address of the current function frame (or process frame if p currently owns no function frame) (section 4.2.4)
- *forallLevel*: *forall* nesting level of this process (section 4.5.3)
- *numForallChildren*: number of p's *forall* children that are currently running (sections 4.5.3 and 4.5.4)

- *maxForallTermiTime*: the most recent termination time of p's *forall* children (section 4.5.4)
- *forallIdxAdr*: while p is executing a *forall* loop, field *forallIdxAdr* contains the memory address of the *forall* index variable (section 4.5.3).
- *repeatProcInGroup*: flag used for implementing the *grouping* option of a *forall* loop (section 4.5.3)
- *forkCount*: number of p's *fork* children that are currently running (sections 4.4.2 and 4.4.3)
- *maxForkTermiTime*: the most recent termination time of p's *fork* children (section 4.4.3)
- *joinCount*: number of p's *fork* children that terminated but have not been matched with a *join* statement yet (section 4.4.4)
- *time*: p's local clock (section 4.3.4)
- *wakeTime*: p's wakeup time (section 4.3.4). When p is put to sleep and the wakeup time is known, p's state is set to *Delayed* and field *wakeTime* records the wakeup time.
- *virProcessor*: ID of the virtual processor on which p is running (section 4.1)
- *phyProcessor*: ID of the physical processor on which p is running (section 4.1)
- *altPhyProsor*: when p evaluates parameters on behalf of its parent, p's actual physical processor ID is temporarily stored in this field, and p's field *phyProcessor* is set to that of the parent. When p completes parameter evaluation, p restores its real physical processor ID from field *altPhyProsor* (section 4.4.2).
- *readChannStatus*: if p is in the middle of a channel read, this field is set to *AtChannel*. When the channel read is successfully completed, *readChannStatus* is reset to *None*, meaning that the process is not reading any channel (section 4.6.6). Field *readChannStatus* can take another value, *TimeReserved*. This value, however, is used only in the simulation of shared-memory architectures.

The data structure of PCB in C language is shown in Figure 11.

List of Processes

When a new process is created, a PCB is allocated and appended to the list of processes. 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 *curProc* pointing to the PCB of the process currently running.

4.3.2 Parallelism by Time Slicing

Parallel execution of application processes are 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 each quantum, the scheduler traverses the list of processes, and schedules every process in a round-robin fashion. If the process is able to run (i.e. it is not blocked or delayed), it executes until its time slice of q time units expires or is put to sleep by some event. The scheduler then gets the next process in the list and schedules this process. As the scheduler moves downward the list, the value of pointer *curProcess* is updated to identify the process currently running. When the last process in the list (i.e. the process whose entry is pointed to by *actProcTail*) finishes its time slice, the global clock (*globClock*) 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 of all processes on a real parallel machine.

The duration of a quantum is the time for a non-header flit to move from one node to an adjacent node (*flit latency*). For example, if $q = 3$, the CEM (code execution module) runs parallel processes for 3 clock cycles, and then the network simulator moves unblocked flits forward by one link. The computation quantum and the communication step are considered to be running in parallel. The quantum can be a fractional number. For instance, when $q = 0.5$, parallel processes run for one clock cycle every time the network simulator advances unblocked flits by two hops.

4.3.3 Process States

Possible states of a process p are

- *Ready*: p can be scheduled for running.

```
typedef struct ProcDescStruct
{
  int processID; /*process ID*/
  int PC; /*program counter*/
  enum States state; /*process state*/
  enum Priority priority; /*scheduling priority*/
  struct ProcDescStruct *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/process frame*/
  int forallLevel; /*forall nesting level of this process*/
  int numForallChildren; /*number of forall chidren 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 chidren 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 altPhyProsor;
      /*to save actual physical processor ID on parameter evaluation*/
  enum ReadStatus readChannStatus; /*status during channel read*/
} ProcDescriptor, *ProcDesPtr;
```

Figure 11: Process control block structure in C

- *Running*: p is currently executing its code.
- *Delayed*: p is put to sleep and the wakeup time is known. The process will be waken up by the scheduler when the wakeup time expires.
- *Blocked*: p is put to sleep and the wakeup time is not known in advance. p will be unblocked by another process (e.g., one of its children) or an event (e.g., the arrival of a message in the wormhole-routed network).
- *Terminated*: p has completed the execution of its code.

Figure 12 shows the interchanges between the states. Following are the conditions for process state transitions.

- from *Delayed* to *Ready*: the wakeup time of p has expired. The state change will be done by the scheduler.
- from *Delayed* to *Running*: p has just been created. Now it is running to evaluate the parameters on behalf of its parent (the parameters are to be passed to the child by the parent). The parent will change p's state from *Delayed* to *Running* during process creation procedure if there are parameters to be passed.
- from *Ready* to *Running*: p is scheduled to run by the scheduler.
- from *Running* to *Blocked*: one of the following cases may cause the state of process to change from *Running* to *Blocked*.
  - p must wait for its newly created children to complete parameter evaluation before proceeding to the next instruction.
  - p must wait for all its *forall* children to terminate before proceeding to the next instruction.
  - p must wait for one or more *fork* children to terminate before it can terminate.
  - p hits a *join* statement with $joinCount = 0$.
  - p must wait on a channel read because no data is currently available for reading.

- from *Running* to *Delayed*: when p has finished parameter evaluation (on behalf of its parent), its state is changed from *Running* to *Delayed.* The scheduler will change p's state to *Running* again when p is scheduled to run.

- from *Running* to *Terminated*: p has completed the execution of its code and has no children to wait for. Therefore p can terminate now.

- from *Blocked* to *Delayed*: one of the following cases may cause the state of process to change from *Blocked* to *Delayed.*

  - p was waiting for its *forall* children to terminate. These children have terminated. Thus p is unblocked.

  - p was waiting for its *fork* children to finish in order to terminate. The last *fork* child has just terminated. Thus p is unblocked. However, the local clock of this process has not reached the termination time of the child (the wakeup time). Therefore, this process is delayed until the wakeup time expires.

  - p was blocked on a *join* and a *fork* child has just terminated. However, the local clock of this process has not reached the termination time of the child (the wakeup time). Therefore, this process is dclayed until the wakeup time expires.

  - p was waiting on a channel read. A message has just arrived and p is the first process in the waiting list at the channel. Thus this process is given the data item.

  In the above cases, the scheduler will change the state of p from *Delayed* to *Running* when p is scheduled to run.

- from *Blocked* to *Running*: p just created one or more child processes. All its new children have completed parameter evaluation on its behalf. This process can now be scheduled to run again.

- from *Blocked* to *Ready*: one of the following cases may cause the state of process to change from *Blocked* to *Ready.*

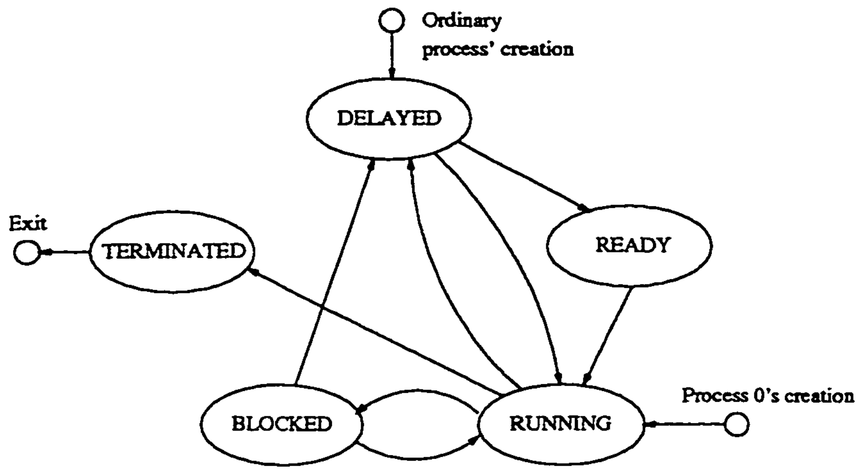


Figure 12: Process state transitions

- p was waiting for its *fork* children to finish in order to terminate. The last *fork* child has just terminated, and the termination time has passed the time on the local clock of p. Thus p is ready to run again.
- p was blocked on a *join* and a *fork* child has just terminated. The child termination time has passed the time on the local clock of p.

4.3.4 Process Scheduling

Every process has its own local clock (field *time* in the PCB). This local clock does not exist in a real system. However, in the simulation, the local clock is needed for the process to synchronize with the global clock and with other processes.

When a process is created, its local clock is initialized with the time on the parent's clock. Every time the process finishes a vCode instruction, its local clock is incremented by the cycle count of that 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 schedule another process to run. The local clock of this process will then try to catch up the time on the global clock.

The scheduler first advances the global clock to the next quantum. It then attempts to schedule every process for running. If the state of a process p is

- *Delayed*, this means that p was put to sleep and the wakeup time is known. If the global clock has not reached the wakeup time ($wakeupTime > globClock$), p continues to sleep. Its local clock is updated to the time on the global clock. If the wakeup time has come ($wakeupTime \leq globClock$), its local clock is set to the wakeup time. The scheduler also changes the process state to *Ready* and then schedules p as a *Ready* process.

- *Running*, the scheduler checks the local clock. If the local clock has not reached the time on the global clock ($time < globClock$), the process is allowed to run until its time slice expires. Thus p is scheduled to run.

  The cycle counts of some instructions may be longer than the quantum duration. It may happen that after an instruction is completed, the local clock of the process exceeds the global clock (i.e., $time \geq globClock$). Every time the scheduler sees $time \geq globClock$, it simply skips p and schedules the next process in the list. p will be able to run again when the global clock surpasses p's local clock (i.e. when $globClock > time$).

- *Ready*, this means that the local clock has not reached the global clock ($time < globClock$), and this is true for all processes with *Ready* state.

  - If p's processor currently has no *Running* process or p is the only process on this processor , p's state is changed to *Running*, and p is scheduled to run.
  - If another process m is *Running* on the same processor, then
    * If m's context switching limit (*switchLimit*) has not expired yet, p is not allowed to run (since m will be using the processor in this quantum).
    * If m has higher priority than p, p is not allowed to run either (even when m's context switching limit has expired).
    * If m's context switching limit has expired and m's priority is not higher than p's, the scheduler performs a context switching on p's processor: p is scheduled to run and its state is updated to *Running*.

  In any case, if p is not allowed to execute in this quantum, p's local clock is updated to the time on the global clock.

- *Blocked*, p's local clock is updated to the time on the global clock.
- *Terminated*, p's entry is removed from the list of processes and its PCB is freed.

The process scheduling algorithm is summarized in Figure 13.

In summary, when the scheduler allows a process to run, the process state is updated to *Running*. When a process starts executing an instruction, this implies that the value on its local clock is less than the value on the global clock.

If the scheduler does not permit a process p to execute in this quantum, and p's local clock has not reached the global clock, then the local clock is updated to the time on the global clock.

If p's state is *Blocked*, an event will unblock p later. Let the time at which the event ends be t, and the value of p's local clock when the event occurs be $time_p$. The event will change p's state to either Ready or Delayed depending on the values of t and $time_p$.

- If $t > time_p$, the process state is changed to *Delayed*, and p's wakeup time is set to t. When p's wakeup time expires, the scheduler will let p run again.
- If $t \leq time_p$, this means that the event had ended and p is thus able to execute now. The event sets p's state to *Ready*.

An exception is when a new child process completes parameter evaluation on behalf of its parent and unblocks the parent. In this case the parent's state is changed from *Blocked* to *Running*. The reason for this exception and more details on parameter evaluation upon process creation will be presented in section 4.4.2.

4.3.5 Deadlock

Two kinds of deadlock may occur during execution of a parallel program: *network deadlock* and *logic deadlock*. Network deadlock may happen due to network resource contention. Messages involved in the deadlock cycle cannot move forward.

Logic deadlock is caused by errors in program logic. For example, the total number of channel reads is larger than the total number of channel writes. In this case, some readers will be blocked forever. When a logic deadlock happens, the state of all existing processes are *Blocked*. However, when the state of all processes are *Blocked*,

```
JobScheduler()
{ logic_deadlock = TRUE;  done = FALSE;  count = 0;
  do { count ++;
   if (curProc == actProcHead) globClock += quantumDuration;
     /*completed one round, so advance global clock to next quantum*/
   p = curProc;  /*p points to PCB of process currently running*/
   if (p->state == Delayed)
     if (p->waketime < globClock) /*time to wake up*/
     { p->state = Ready;
       if (p->waketime > p->time) /*wakeup time exceeds local clock*/
         p->time = p->waketime;   /*update p's local clock*/
     } else                       /*continue to sleep*/
     { p->time = globClock;       /*update p's local clock*/
       logic_deadlock = FALSE;
     }
   if (p->state == Running)
   { logic_deadlock = FALSE;
     if (p->time < globClock) done = TRUE; /*p will continue to run*/
   } else if (p->state == Ready)
   { logic_deadlock = FALSE;
     m = pointer to PCB of process currently Running on p's processor;
     if (m == NULL) /*there is no Running process on p's processor*/
     { curProc = p;  p->state = Running; /*p is scheduled to run*/
       done = TRUE;
     } else if ((m reached contextSwitchLimit) and
             (m's priority <= p's priority)) /*then context-switch*/
     { curProc = p;  p->state = Running;
       m->state = Ready;  done = TRUE;
     } else  p->time = globClock;   /*p is not allowed to run*/
   } else if (p->state == Blocked)  /*continue to sleep*/
      p->time = globClock;          /*update p's local clock*/
   else if (p->state == Terminated)   /*p terminated*/
      remove p's PCB from the list of processes;
   else if (! done)  /*if p gets here, p is not allowed to run*/
      p = p->next;   /*consider the next process in the list*/
  } while(!(done or (logic_deadlock and
                    count > total number of processes)));
  if (logic_deadlock and no messages in network) System_Deadlock();
}
```

Figure 13: Process scheduling algorithm

this may not mean a logic deadlock. In this case, the reason may be that all processes are waiting for messages that are being routed and have not arrived at the destinations yet.

A *system deadlock* that really blocks program execution happens when one of the following scenarios exists:

- There is a network deadlock. This implies that some message cannot be delivered to the destination. Therefore, its intended reader process would be blocked forever.
- The state of all processes are *Blocked*, and there are no messages to be routed in the network. This implies a real logic deadlock.

When a system deadlock happens, the program execution is aborted. Detailed information about the deadlock can be obtained from the debugging monitor.

a) Body is a function call with parameter passing
```
fork (i+1;) Add(op1, op2, &sum);   /*map to virtual processor (i+1)*/
```

b) Body is a function call with no parameter passing
```
fork ( ; chan1, chan2) PrintResult();   /*owner of 2 channel variables*/
```

c) Body is a block statement
```
fork {
    int k;
    Input(k);
      ...
    }
```

d) Body is a single statement
```
fork printf("Hello world!");
```

Figure 14: Examples with *fork* statement

4.4 *fork* Processes

4.4.1 *fork* Statement

Execution of a *fork* statement will create a new process. The parent can specify the virtual processor on which the child process will run. The parent can also assign channel variables to the child so that the child will use these channel variables to communicate with other processes. The child will be the owner of the assigned channel variables and only it can read from these channels.

After a parent process spawns a *fork* child, it can continue with the instruction following the *fork* right away. The parent and the child are then running in parallel.

When a parent process wants to terminate, it must wait for all of its children to finish first. The parent will be blocked until the last child terminates; this child will wake up the parent and let the parent terminate.

Detailed description of the syntax and semantics of *fork* statement is provided in Appendix A. Figure 14 provides examples of *fork* which will be used for discussion in this section.

```
Step 1 : Calculate the ID of the virtual processor on which the child will run;
Step 2 : Assign the specified channel variable(s) to the child;
Step 3 : **NewForkChild**; /*create a new fork child*/
Step 4 : **ForkJump**; /*after creating the child, the parent jumps to
                        the next instruction following the fork (Step 5b)*/
Step 5a: Child executes the code body of the fork;
Step 6 : **ForkChildEnd**; /*the fork child terminates*/
Step 5b: The parent executes the instruction following the fork
```

Figure 15: Algorithm for *fork* process creation (vCode instructions are in bold face)

4.4.2 *fork* Process Creation

The algorithm for executing a *fork* is shown in Figure 15. In the algorithm, the parent process executes steps 1, 2, 3, 4, and 5b, while the new *fork* child will execute steps 5a and 6.

Parent's Operations

The parent process first computes the ID of the virtual processor on which the new child will run (Step 1 in Figure 15). If the user specifies the virtual processor ID (as in Example *a* in Figure 14), the parent evaluates the value of the expression denoting the processor ID. In the mentioned example, the virtual processor ID is the value of the expression $i + 1$. If no processor ID is specified, the parent provides a default virtual processor ID by executing vCode instruction *DefaultProc*. The algorithm of instruction *DefaultProc* was described in section 4.1.4. In any case, the child's virtual processor ID is pushed onto the parent's stack. The virtual processor will be mapped to a physical processor later using the mapping table.

If there are channel variables to assign to the child (Example *b*, Figure 14), the parent binds these variables to the child's processor (Step 2 in Figure 15). vCode instruction *MVChannVar* is executed for each channel variable to be bound. Let *a* be the address in the memory pool of a channel variable to be assigned, *MVChannVar* sets `storageOwner[a]` to the child's physical processor ID to make the new child the owner of this channel variable (i.e. only the child is allowed to read from this channel).

The parent then executes instruction *NewForkChild* to really create a new process (Step 3 in Figure 15). The algorithm of *NewForkChild* is discussed below and summarized in Figure 16.

- The parent first allocates a PCB to the new child.
- The new PCB is initialized. Function *Init_PCB*() in Figure 17 contains the initialization code.
- The new PCB is appended to the end of the list of processes.
- The parent requests a process frame for the new child. The pointers *base*, *stackTopLim* and *T* (in the child's PCB) are set to keep track of the allocated process frame.
- The child's virtual processor ID on top of the parent's stack is mapped to a physical processor ID using the mapping table. Using the physical processor ID, the parent updates the corresponding physical processor descriptor. The processor status is set to *Occupied* and the number of processes using this processor is incremented by one to include the new child.
- The child's virtual processor ID on top of the parent's stack is no longer needed. Thus it is popped off the stack.
- The parent copies its *forall information block* (FIB), if any, to the new child's working stack. The structure and use of FIBs will be discussed in section 4.5.2.
- Field *forkCount* in the parent's PCB is incremented by one to account for the new *fork* child.
- If no parameter passing from the parent to the child is required (Examples *b*, *c*, and *d* in Figure 14), the parent sends a birth message to the child's processor. However, if the child's code is a function call with parameter passing (Example *a* in Figure 14), the birth message will be combined with the parameters into one message which will be sent to the child's physical processor after the parameters have been evaluated. The parent will let the child evaluate the parameters on its behalf. Therefore it executes function *Prepare_Parm_Eval*() to prepare the child for this task. The code of function *Prepare_Parm_Eval*() is shown in Figure 18.

After the parent has finished the execution of *NewForkChild*, the child becomes an independent process and will be scheduled for running. If the *fork* does not involve any parameter passing as in Example *d* in Figure 14, the parent immediately proceeds to the next instruction that is *ForkJump* (Step 4 in Figure 15). All *ForkJump* does is to set the parent's program counter to the instruction following the *fork* statement in the CPC program. The parent then continues its execution in parallel with the child (Step 5b in Figure 15).

If the *fork* is accompanied by a function call with parameter passing, the parent would have to evaluate the parameters and send the values of the parameters to the child. However, in our implementation, the child will evaluate the parameters on behalf of the parent, and charge the processing time to the parent's processor. The reason for this implementation will be explained shortly. During the parameter evaluation, the parent process is blocked. When the child finishes evaluating parameters, it wakes up the parent. The parent resumes execution with instruction *ForkJump* as just mentioned above (Step 4 and 5b in Figure 15).

Child's Operations

If the *fork* does not involve any parameter passing (Examples *b*, *c* and *d* in Figure 14), the child simply starts executing its own code (Step 5a in Figure 15) and runs in parallel with the parent. Consider Example *d* in Figure 14. After the parent finishes instruction *NewForkChild* and the birth message arrives at the child's processor, the *fork* child executes *printf* statement and then vCode instruction *ForkChildEnd* to terminate. In Example *b* (or *c*) in Figure 14, the new *fork* child calls a function (or pseudo function) with no parameter passing from the parent. In this case, the child runs the code for a function call as shown in Figure 10.

However if the *fork* requires parameter passing from the parent to the child, the child begins by evaluating the parameters on behalf of the parent. On a real multicomputer, the parent would evaluate the parameters and pass them to the new child. We could simulate the same thing: the parent would evaluate and save the parameters on its stack. The parent would then copy the final values of the parameters to the child's stack. To eliminate this copying and thus reduce simulation time, we let the new child evaluate the parameters. The final values of the parameters are thus stored directly on the child's stack. The following details must also be included in

```
NewForkChild
{
  /*Precondition: ID of child's virtual processor is on top of the parent's stack*/
  /*childPtr: pointer to the PCB of the new fork child*/
  /*parentPtr: pointer to the PCB of the parent
                              (process running this NewForkChild)*/
  /*childPhysProcID: ID of child's physical processor*/

  Allocate a PCB for the new child;
  Init_PCB(); /*initialize the child's PCB (Figure 17)*/
  Append the new PCB to the list of processes;
  Allocate a process frame for the child {
    childPtr->base = starting address of the process frame;
    childPtr->T = childPtr->base - 1;
    childPtr->stackTopLim = childPtr->base +
                    (size of process frame) - 1; }
  Update the descriptor of the child's processor {
    physProsorTable[childPhysProcID].status = Used;
    physProsorTable[childPhysProcID].nbrProcesses++; }
  Pop the top value off the parent's stack;
   /*this value is child's virtual processor ID that was used in function Init_PCB()*/
  CopyPreviousForallInfo();
    /*copy previous forall info from parent's FIB to child's process frame*/
    /*forall information block will be explained in section 4.5.2*/
  parentPtr->forkCount++; /*parent has one more fork child*/
  if the child begins with a function call /*e.g. Example a, Figure 14*/
    Prepare_Parm_Eval(); /*prepare for parameter evaluation (Figure 18)*/
  else   /*child's code is a single statement (e.g. Example d, Figure 14)*/
    Send a birth message to the child's processor;
}
```

Figure 16: Algorithm of vCode instruction *NewForkChild*

```
Init_PCB()
{
 childPtr->processID = the next available process ID;
 childPtr->PC = parentPtr->PC + 1;
     /*vCode generation ensures that the first instruction of the
       child's code is always at (parentPtr->PC + 1)*/
 childPtr->state = Blocked; /*until the birth message arrives*/
 childPtr->priority = LowPri;
 childPtr->parent = parentPtr;
 childPtr->B = parentPtr->B;
 childPtr->forallLevel = parentPtr->forallLevel;
 childPtr->numForallChildren = 0;
 childPtr->maxForallChildTime = 0.0;
 childPtr->repeatProcInGroup = FALSE;
 childPtr->forkCount = 1; /*child counts itself as one fork child*/
 childPtr->maxForkTermiTime = 0.0;
 childPtr->joinCount = 0;
 childPtr->time = parentPtr->time;
 childPtr->virtualTime = 0.0; /*child just started*/
 childPtr->virProcessor = value on top of the parent's stack;
 childPtr->phyProcessor = physical processor ID; /*from mapping table*/
 childPtr->altPhyProsor = Nil;
 childPtr->readChannStatus = None;
}
```

Figure 17: Initializing a PCB

```
Prepare_Parm_Eval()
{
 /*parentProcID: ID of the parent's processor*/
 parentPtr->state = Blocked; /*until child finishes parameter evaluation*/
 childPtr->state = Running; /*to evaluate parameters*/
 childPtr->priority = HighPri;
 childPtr->altPhyProsor = childPtr->physProcessor; /*child is temporarily
                    using parent's processor, so save child's processor ID*/
 childPtr->physProcessor = parentPtr->physProcessor; /*= parentProcID*/
       /*parameter evaluation time is charged to the parent's processor*/
 physProsorTable[parentProcID].runProcess = childPtr; /*child is running*/
 physProsorTable[parentProcID].nbrProcesses++; /*counting the child*/
}
```

Figure 18: Parent process preparing for parameter evaluation

this implementation.

- We must charge parameter evaluation time to the parent's processor and not to the child's processor because parameter evaluation is actually the parent's job. This is why in function *Prepare_Parm_Eval* (Figure 18) we save the ID of the child processor in field *altPhyProsor*, and then set the child's processor to that of the parent.

- While the child is evaluating the parameters, the parent must be blocked. When the child completes parameter evaluation, it wakes up the parent so that the parent can resume its execution.

In summary, if the *fork* is followed by a function call with parameters, the child will execute the code shown in Figure 19. Compared with the code sequence of an ordinary function call in Figure 10, the code sequence in Figure 19 is added with an extra vCode instruction, which is *WakeupProcess*. This instruction lets the child wake up the parent after the parameters are evaluated. The pseudo-code of instruction *WakeupProcess* is given in Figure 20. After waking up the parent, the child calls the function as any other ordinary function by executing instruction *Call*.

```
**NewFrame**; /*allocate a function frame for the function*/
Evaluate the parameters;
**WakeupProcess**; /*wake up parent after parameter evaluation*/
**Call**;   /*call the function*/
```

Figure 19: *fork* with parameter passing (vCode instructions are in bold face)

```
WakeupProcess
{
 /*childPtr: pointer to the PCB of the fork child that wakes up the parent*/
 /*parentPtr: pointer to the PCB of the parent*/
 /*parentProcID: ID of the parent's physical processor*/

 if (!child->repeatProcInGroup)
  /*repeatProcInGroup is used for forall process creation (section 4.5.3).
    repeatProcInGroup is always set to FALSE for a fork*/
 { parentPtr->time = childPtr->time
   parentPtr->state = Running;
   physProsorTable[parentProcID].runProcess = parentPtr;
                       /*parent resumes execution on this processor*/
   physProsorTable[parentProcID].nbrProcesses--;
                      /*remove the child from the parent's processor*/
   childPtr->physProcessor = childPtr->altPhyProsor;
                      /*move the child back to its actual processor*/
   childPtr->priority = LowPri;
   Send birth message and parameters from parent's processor to child's processor;
   childPtr->state = Blocked; /*until birth message and parameters arrive*/
 }
}
```

Figure 20: Algorithm of vCode instruction *WakeupProcess*

4.4.3 *fork* Process Termination

After a *fork* child has completed its code, it executes vCode instruction *ForkChildEnd* to terminate. The algorithm of instruction *ForkChildEnd* is discussed below and summarized in Figure 21.

The terminating *fork* process p itself may be the parent of other *fork* children. If this is the case, p must wait for all of its *fork* children to terminate before it can terminate. p checks field *forkCount* in its PCB. If $forkCount > 1$, p has at least one *fork* child that is still running. Thus p runs the following steps:

- p decrements *forkCount* by one (because when p was created, it counted itself as one *fork* child of its own).
- p decrements its program counter by one so that when p's last child has finished, p will re-run this instruction *ForkChildEnd* to terminate.
- p releases the processor on which it is running.
- p's state is set to *Blocked.* p is put to sleep until p's last *fork* child terminates and unblocks p.

If all p's *fork* children have terminated, p executes the following steps to terminate itself.

- p returns its process frame to the memory pool.
- p sends a death message to its parent, say process q, to notify q that it has terminated.
- p updates q's state and child information by calling *forkDeathProcessing*(). This function will be described shortly.
- p releases the processor on which it is running. p also updates the processor descriptor accordingly.
- p's state is set to *Terminated.* The scheduler will remove p's PCB later when it attempts to schedule p and sees p's state is *Terminated.*

```
ForkChildEnd
{
 /* p: pointer to PCB of process executing this instruction */
 if (p->forkCount > 1)
 { p->forkCount--;
   p->PC--; /*p reruns this instruction when the last child finishes*/
   Release the processor on which p is running;
   p->state = Blocked;
 }
 else /*all fork children terminated*/
 { Return process frame to the memory pool;
   Send a death message to the parent;
   if (death message is intra-processor) /*arrival time is known*/
     forkDeathProcessing(); /*update parent's state and child information*/
   Release the processor on which p is running;
   p->state = Terminated;
 }
}
```

Figure 21: Algorithm of vCode instruction *ForkChildEnd*

Function *forkDeathProcessing*() updates the process state and child information of the parent process q. The pseudo-code is shown in Figure 22, and the algorithm is as follows.

- Field *forkCount* of q's PCB is decremented.
- Field *maxForkTermiTime* of q's PCB is updated (if required).
- If q is being blocked on a *join*, q is unblocked so that q will be able to run again. Field *joinCount* of q's PCB is incremented to account for p's termination.
- If q is blocked on its termination and p is the last child that terminates, q is unblocked so that q will be able to terminate as well.

We now explain why a new *fork* process counts itself as one *fork* child of its own (Figure 17). A process p can be blocked for several reasons. One of the reason is that p is the parent of one or more *fork* children and wants to terminate, but at least one *fork* child of p is still running. Thus p's state should be set to *TerminatingWaiting*

```
forkDeathProcessing()
{
  /* arrivalTime: arrival time of p's death message */
  /* q: pointer to PCB of the parent of terminating child p */

  q->forkCount--; /*one less fork child*/
  if (q->maxForkTermiTime < arrivalTime)
    q->maxForkTermiTime = arrivalTime;
  if (q->joinCount == -1)  /*q is currently blocked on a join*/
    if(q->time < arrivalTime)
      {q->wakeTime = arrivalTime; q->state = Delayed;}
    else q->state = Ready;
  q->joinCount++; /*one more fork child terminates*/
  if (q->forkCount == 0)
          /*q terminated and the last child (p) is terminating*/
    if (q->time < q->maxForkTermiTime)
      { q->state = Delayed;
        q->wakeTime = q->maxForkTermiTime;
      }
    else q->state = Ready;
}
```

Figure 22: Function *forkDeathProcessing*()

(terminating and waiting for children to finish first). When a *fork* child of p terminates, it checks if (parent's state = *TerminatedWaiting* and parent's *forkCount* = 0) is true. If so, it should unblock the parent to let the parent terminate. We wanted to eliminate one process state (*TerminatingWaiting*) and one of the two conditions in the above *if* statement. Thus we let p count itself as one *fork* child of its own when it is created. When p wants to terminate, it decrements *forkCount* by one for itself. When a *fork* child f of p sees the parent's forkCount = 0, f knows that its parent has terminated and is waiting for f to terminate. This is equivalent to the previous *if* statement: if (parent's state = *TerminatedWaiting* and parent's *forkCount* = 0).

4.4.4 *join* Statement

An example of the *join* statement is shown below. More explanations on the use of *join* statement can be found in Appendix A.

```
...
for(i = 0; i < 10; i++)
   fork (i) Compute(i);
...
for(i = 0; i < 10; i++)
   join;
...
```

To implement the *join* statement, each process needs a variable called *joinCount* which is stored in the PCB. *joinCount* is initialized to 0. Whenever a *fork* child terminates, *joinCount* is incremented by 1. Thus *joinCount* of a process records the number of *fork* children of that process which have terminated. *joinCount* can be thought of as the complement of *forkCount*. But there is more to *joinCount* as explained below.

When a process hits a *join* statement, its *joinCount* is checked. If $joinCount > 0$, this means that one or more *fork* children have terminated. So the process passes this *join* to get to the next instruction. It also decrements *joinCount* by 1. More precisely, *joinCount* is the number of *fork* children which have terminated and not been matched with a *join* yet.

If $joinCount = 0$, the process is then blocked by this *join*. *joinCount* is set to -1 to indicate that the process has an outstanding *join* and is waiting for a *fork* child to terminate in order to pass this *join*.

The process is blocked until one of its child terminates by executing *ForkChildEnd*. In *ForkChildEnd*, the child increments its parent's *joinCount* and unblocks the parent (Figure 21). The parent can then pass the *join* and resume execution.

A *join* statement in the CPC program is translated to vCode instruction *join* whose implementation is shown below.

join

```
{
  /* p: pointer to the PCB of the process executing this join */
  if (p->joinCount > 0)
     p->joinCount--; /*pass this join*/
  else
  { p->joinCount = -1
    p->state = Blocked; /*until a fork child terminates*/
    physProsorTable[p->physProcessor].runProcess = NULL;
                /*release the processor while being blocked*/
  }
}
```

*/*Example a*/*

```
forall i (0; 10; 4)     /*from 0 to 10, grouping 4*/
   (i/4; C[i]) Compute(i);
```

*/*Example b*/*

```
forall i (0; 10; 3)     /*from 0 to 10, grouping 3*/
  forall j (1; 20; 5)   /*from 1 to 20, grouping 5*/
    forall k (0; 4; )   /*from 0 to  4, grouping 1 (default)*/
      Calculate(i, j, k);
```

*/*Example c*/*

```
forall i (0; 10; )      /*from 0 to 10, grouping 1 (default)*/
  PrintOutput();
```

Figure 23: Examples of *forall* statement

4.5 *forall* Processes

4.5.1 *forall* Statement

Execution of a *forall* statement allows a parent process to create one or more *forall* children. The number of *forall* children to be created is determined by the values of the lower bound and the upper bound of the *forall* index, and the value of the *grouping*. In Example *a* in Figure 23, three child processes are created: the first process will execute function *Compute*() for i from 0 to 3; the second process, from 4 to 7; and the third process, from 8 to 10.

For each *forall* child to be created, the user can specify the virtual processor on which the new child will run. The parent process can also bind the channel variables to the child process so that the child will communicate with other processes using these channel variables. The child will receive messages from other processes via the assigned channels, and only it can read from these channels.

After a parent process has created all *forall* children required by the *forall* loop, it will be blocked until all these *forall* children terminate. The last *forall* child that finishes will wake up the parent. Only then can the parent resume execution.

Figure 23 shows examples of *forall* statement, which will be used for discussion in this section. More details on the syntax and semantics of *forall* statement are given in Appendix A.

Following are the definitions of some terms that will be needed for discussing *forall* processes.

- *forall* level: The *forall* level of a process is defined as follows.
  - Process 0 is at *forall* level 0.
  - If the *forall* level of a process p is k, a *forall* child created by p is at *forall* level $k+1$.
  - A *fork* child has the same *forall* level as its parent.
- *from* bound (*fromB*) and *to* bound (*toB*): the lower bound and the upper bound of the *forall* index specified in the *forall* loop, respectively.

  In Example a in Figure 23, the *from* bound and the *to* bound of the *forall* loop are 0 and 10 respectively.
- child lower bound (*childLoB*) and child upper bound (*childUpB*): Each *forall* child is considered to execute a *for* loop whose number of iterations is the value of the *grouping*, say g. If $(toB - fromB + 1)$ is not divisible by g, then the *for* loop of the last child has less than g iterations.

  The lower bound and upper bound of such a *for* loop of a new child is referred to as the *child lower bound* and *child upper bound* respectively. In Example a in Figure 23, the first child has $childLoB = 0$ and $childUpB = 3$; the second child, $childLoB = 4$ and $childUpB = 7$; the third child, $childLoB = 8$ and $childUpB = 10$.
- current loop lower bound (*curLoopLoB*): After a *forall* child has been created, the *forall* index is incremented by the *grouping* value to prepare for the creation of the next child. The current value of the *forall* index is referred to as the *current loop lower bound.* The *curLoopLoB* will become the child lower bound of the next child to be created. The current loop lower bound is used only by the parent process. After creating a *forall* child, the parent compares the current loop lower bound against the *to* bound. If $curLoopLoB > toB$, the parent exits the *forall* loop.

4.5.2 *forall* Information Blocks

The index variable of the *forall* loop is not shared by all the *forall* children. Each child process can see only a unique value of the index. To allow the children to refer to the index, each child process must keep its own copy of the index. Also, each new *forall* child has its own child lower bound and child upper bound. These values must also be recorded in order for the child process to iterate a number of times required by the *grouping*. *forall* information blocks are used to hold all the information just mentioned.

Every process at *forall* level 1 and up has a *forall information block* (FIB) stored in the process frame, starting at the lowest address of the frame. Since *forall* statement can be nested (Example *b* in Figure 23), the FIB of a process at *forall* level k records the information of *forall* loops from level 1 to level k.

When a new process is created at *forall* level 1 (Example *a* in Figure 23), the child's FIB contains the following information.

```
/*child's forall level == 1*/
/*child: pointer to the new child's PCB*/
/*base = child->base*/
storageValue[base].val = childLoB;  /*child lower bound*/
storageOwner[base] = address of forall index at level 1;
storageValue[base+1].val = childUpB;  /*child upper bound*/
/*storageOwner[base+1] is unused*/
```

Note that in a FIB, the entries of array `storageOwner[]` no longer hold processor IDs but store addresses of *forall* indexes at different levels. The FIB of a *forall* child at level 1 is depicted in Figure 24a.

When a parent at forall level k creates a new child, it copies most information in its FIB to the child's process frame. More specifically, the first k words of the parent's FIB are copied to the child's FIB using function *CopyPreviousForallInfo*().

```
/*parent's forall level == k (k >= 0)*/
/*parent: pointer to the parent's PCB*/
/*child: pointer to the new child's PCB*/
/*baseChild  = child->base*/
/*baseParent = parent->base*/
```

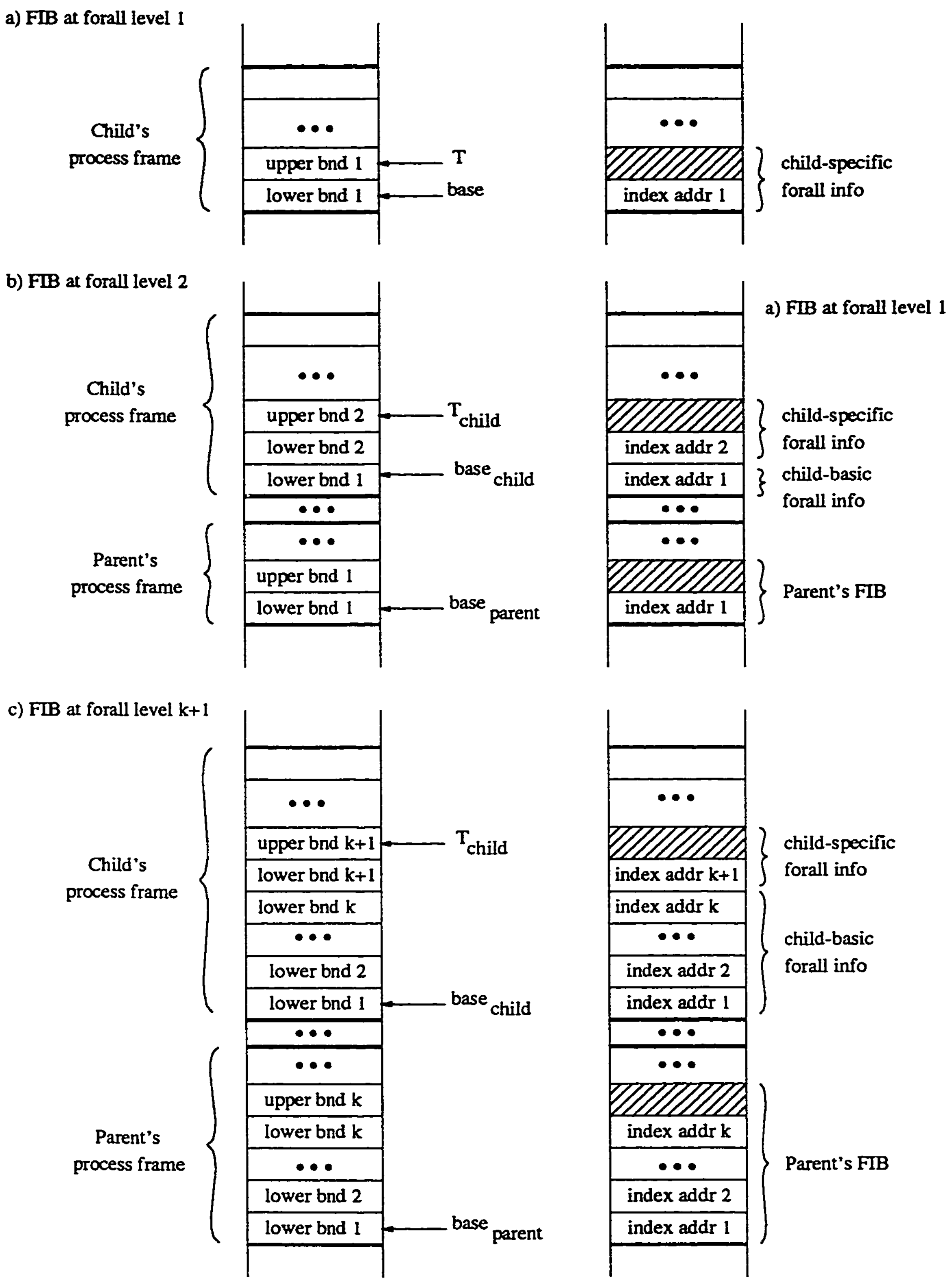


Figure 24: Structure of *forall* information blocks

```
CopyPreviousForallInfo()
{
   for(x = 0; x <= k-1; x++)
   {
      storageValue[baseChild+x].val = storageValue[baseParent+x].val;
                     /*child lower bounds from level 1 to k*/
      storageOwner[baseChild+x]     = storageOwner[baseParent+x];
                     /*addresses of forall indexes at level 1 to k*/
   }
}
```

This block of k words is called *previous forall information.* All children at level $k+1$ of a process p will have the same k words of forall information copied from p's FIB.

Assume that the parent's *forall* level is k ($k \geq 0$). If the new child is a *fork* child, the child's forall level is also k (i.e., `child->forallLevel = k`). The child's FIB holds all the previous forall information copied from the parent's FIB as described above.

If the new child is a *forall* child, the child's forall level is now $k+1$ (i.e., `child->forallLevel = k+1`). Since this is a *forall* child, the parent will add more information related to the child's own *forall* loop to the child's FIB. This is done by calling function $AddCurrentForallInfo()$.

```
AddCurrentForallInfo()
{
   storageValue[baseChild+k].val = childLoB;   /*child lower bound*/
   storageOwner[baseChild+k] = address of forall index at level k+1;
   storageValue[baseChild+k+1].val = childUpB; /*child upper bound*/
   /*storageOwner[baseChild+k+1] is unused*/
}
```

This added information is called *current forall information* since each distinct child at level $k+1$ has its own child lower bound and child upper bound. Figure 24b and c illustrates the FIB of a *forall* child at level 2 and $k+1$, respectively.

```
Step  1: Load the address of the *forall* index onto the stack top;
Step  2: Evaluate the *from* bound and push the final value onto the stack top;
Step  3: Evaluate the *to* bound and push the final value onto the stack top;
Step  4: Evaluate the *grouping* expression and push the final value
          onto the stack top;
Step  5: **BegParallel**;
Step  6: **BegForallLoop**; /*jump to Step 14 when done*/
Step  7: Evaluate the virtual processor ID of the child (if specified);
          Push the child's physical processor ID onto the parent's stack top;
Step  8: Bind the specified channel variable(s) to the child;
Step  9: **NewForallChild**; /*create a new forall child*/
Step 10: **Jump**; /*jump back to instruction BegForallLoop in Step 6*/
Step 11: Child executes the code body of the *forall*;
Step 12: **TstGrpIncIdx**; /*check grouping option; increment forall index*/
Step 13: **ForallChildEnd**; /*the forall child terminates*/
Step 14: **EndForallLoop**;
Step 15: **EndParallel**;
```

Figure 25: The steps of executing a *forall* statement

4.5.3 *forall* Process Creation

The algorithm for executing a *forall* statement is described in Figure 25. The parent process runs steps 1 to 10, and 14 to 15. The new *forall* child will executes steps 11 to 13.

Parent's Operations

The parent first calculates the address of the *forall* index (if necessary), and pushes the address onto the top of its stack (Step 1, Figure 25). In Example *a* in Figure 23, the address of index *i* is loaded onto the parent's stack. The parent then evaluates the *from* bound expression, and pushes the final value onto its stack (Step 2, Figure 25). The same operations are performed for the *to* bound and the *grouping* expression (Step 3 and 4, Figure 25). After Step 4, the parent's stack is as shown in Figure 26a.

During the parent's execution of the *forall* loop, the values of the index address, the *to* bound and the *grouping* are unchanged. The *from* bound, however, becomes the current loop lower bound (*curLoopLoB*), and the *curLoopLoB* is incremented by the *grouping* value after a new child has just been created. The increment is

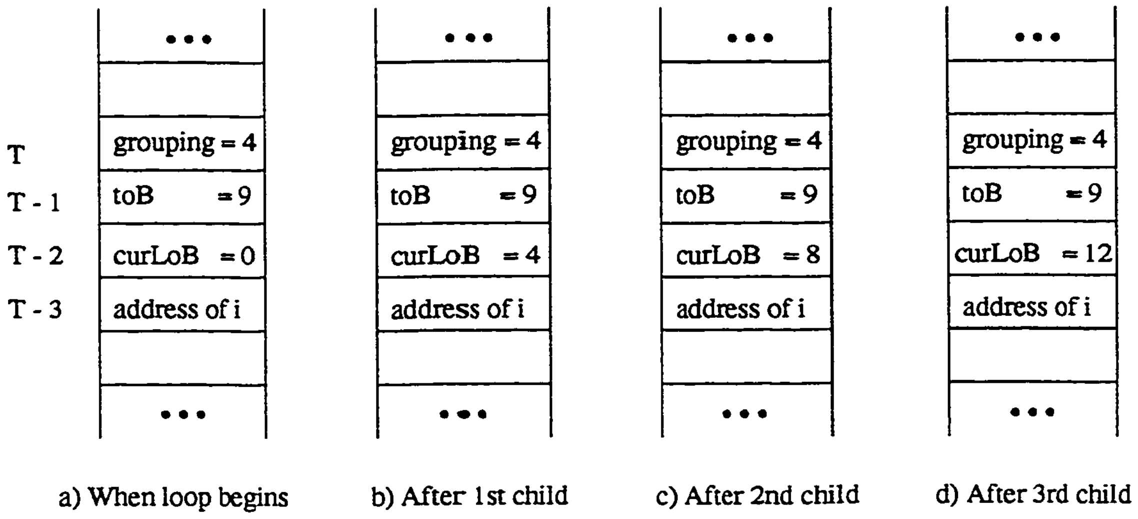


Figure 26: The parent's stack during execution of a *forall* loop

performed by instruction *EndForallLoop* (Step 14, Figure 25) on memory location `storageValue[T-2]` (Figure 26b, c), where T is the current top of the parent's stack.

Right after Step 4, the parent executes vCode instruction *BegParallel* which resets field *numForallChildren* in the parent's PCB to 0 (Step 5, Figure 25). The next instruction to be executed (Step 6, Figure 25) is *BegForallLoop* whose algorithm is as follows.

- The *grouping* value is validated. If *grouping* $\leq$ 0, the system state is set to *GrpErr* (Grouping Error) which will halt the program execution.

- The current loop lower bound (*curLoopLoB*) is compared against the *to* bound. If *curLoopLoB* $>$ *toB*, this means that the parent has created all the children required by the *forall* loop. The parent will pop the top four elements off its stack (which were pushed in during steps 1, 2, 3, and 4). The program counter is set to instruction *EndParallel* (Step 15, Figure 25) so that the parent will execute this instruction next.

  If *curLoopLoB* $\leq$ *toB*, this means that the parent still has one or more children to create. The parent's priority is set to *HighPri* for this task.

Figure 27 shows the pseudo-code of instruction *BegForallLoop*.

```
BegForallLoop
{
 if (grouping <= 0)
   systemState = GrpErr;
 else if (curLoopLoB > toB) /*no children to create*/
 { pop the top four elements off the parent's stack;
   set PC to instruction EndParallel; /*step 15, Figure 25*/
 }
 else set priority to HighPri;
}
```

Figure 27: Instruction *BegForallLoop*

Similar to the case of a *fork* statement, before creating a child, the parent calculates the child's virtual processor ID if it is specified in the *forall* statement. Otherwise, a default physical processor ID is obtained from execution of instruction *DefaultProc* (section 4.1.4). In either case, the child's physical processor ID is loaded on top of the parent's stack. This is done in Step 7 of Figure 25.

If there are channel variables to assign to the child, the parent binds these variables to the child's processor (Step 8, Figure 25) by executing vCode instruction *MVChannVar* for each channel variable to be bound. This step was described in detail in section 4.4.2.

The parent then executes instruction *NewForallChild* to really create a new *forall* process (Step 9, Figure 25). Instruction *NewForallChild* is similar to vCode instruction *NewForkChild*. Their differences are listed below.

- After the previous forall information is copied from the parent's FIB to the child process frame using function *CopyPreviousForallInfo*(), the parent adds current forall information to the child's FIB by calling function *AddCurrentForallInfo*(). These two functions were described in section 4.5.2.

- If this is the parent's first *forall* child, the parent accesses its PCB and updates field *maxForallTermiTime* to the time on its local clock.

  After creating all required *forall* children, the parent process will be blocked until all its *forall* children have terminated. *maxForallTermiTime* records the death time of the *forall* child that terminated the most recently. After the

```
NewForallChild
{
  /*Precondition: ID of child's processor is on top of the parent's stack*/
  /*childPtr: pointer to the PCB of the new fork child*/
  /*parentPtr: pointer to the PCB of the parent
                              (process running this NewForkChild)*/
  /*childProcID: ID of child's processor*/

  Allocate a PCB for the new child;
  Init_PCB(); /*initialize the child's PCB (Figure 17)*/
  Append the new PCB to the list of processes;
  Allocate a workspace for the child {
    childPtr->base = starting address of the workspace;
    childPtr->T = childPtr->base - 1;
    childPtr->stackTopLim = childPtr->base +
                      (size of process frame) - 1; }
  Update the descriptor of the child's processor {
    virProsorTable[childProcID].status = Used;
    virProsorTable[childProcID].nbrProcesses++; }
  Pop the top value off the parent's stack;
   /*this value is child's virtual processor ID that was used in function Init_PCB()*/
  CopyPreviousForallInfo();
     /*copy previous forall info from parent's FIB to child's process frame*/
  AddCurrentForallInfo();
     /*add current forall info to child's FIB*/
  if (parentPtr->numForallChildren == 0)
    parentPtr->maxForallChildTime = parentPtr->time;
  parentPtr->numForallChildren++; /*one more forall child*/
  if the child begins with a function call /*e.g. Figure 14a*/
    Prepare_Parm_Eval(); /*prepare for parameter evaluation (Figure 18)*/
  else    /*child's code is a single statement (e.g. Figure 14d)*/
    Send a birth message to the child's processor;
}
```

Figure 28: Algorithm of vCode instruction *NewForallChild*

last *forall* child terminated, the scheduler checks field *maxForallTermiTime* of the parent to decide whether the parent is allowed to resume execution.

- Field *numForallChildren* (instead of field *forkCount*) in the parent's PCB is incremented by one to account for the new *forall* child.

The algorithm of *NewForallChild* is given in Figure 28.

If the *forall* does not require any parameter passing (Example *c*, Figure 23) after finishing instruction *NewForallChild*, the parent executes instruction *Jump* which sets the parent's program counter to the code address of instruction *EndForallLoop* (Step 14, Figure 25). If the *forall* body is a function call with parameter passing (Example *a*, Figure 23), the parent will let the child evaluate the parameters on its behalf (the reason for this implementation was explained in section 4.4.2). The parent will be blocked until the child finishes parameter evaluation and wakes it up. The parent then resumes execution with instruction *Jump* to jump to instruction *EndForallLoop* as just mentioned above. Following is the algorithm of instruction *EndForallLoop*.

- The current loop lower bound (*curLoopLoB*) is incremented by the *grouping* value.

- The current loop lower bound (*curLoopLoB*) is compared against the *to* bound. If $curLoopLoB \leq toB$, the parent's program counter is set to the code address of instruction *BegForallLoop* (Step 6, Figure 25) so that the parent will repeat the loop to create the next *forall* child.

  If $curLoopLoB > toB$, this means that the parent has created all the children required by the *forall* loop. The parent will pop the top four elements off its stack (which were pushed in in steps 1, 2, 3, and 4). The parent's priority is set back to *LowPri* since it is no longer required to create any more children for this *forall* loop (in the CPSS, process creation is given high priority so that new processes are spawned as early as possible).

The pseudo-code of instruction *EndForallLoop* is given in Figure 29.

In instructions *BegForallLoop* and *EndForallLoop* (Steps 6 and 14 respectively), if $curLoopLoB > toB$, the next instruction to be executed is *EndParallel*. In this instruction, if at least one *forall* child is still running, the parent is blocked until all

```
EndForallLoop
{
 curLoopLoB = curLoopLoB + grouping;
         /*curLoopLoB is stored at storageValue[T-2]*/
 if (curLoopLoB <= toB)
   set parent's PC to instruction BegForallLoop; /*step 6, Figure 25*/
 else
 { pop the top four elements off the stack;
   set parent's priority to LowPri;
 }
}
```

Figure 29: Instruction *EndForallLoop*

of its *forall* children terminated. Only then is the parent allowed to run again. The pseudo-code of instruction *EndParallel* is as follows.

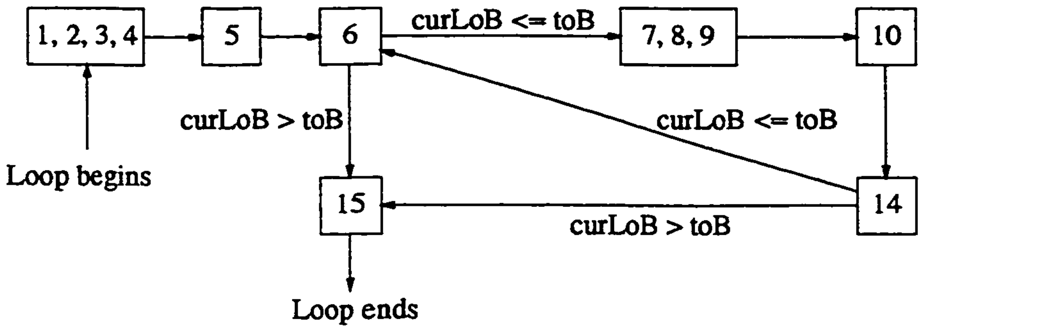
```
EndParallel
{
 /*parentPCB: pointer to the parent's PCB*/
 if (parentPCB->numForallChildren > 0) /*some child still running*/
 { parentPCB->state = Blocked; /*blocked until all children finish*/
   physProsorTable[parentPCB->physProcessor].runProcess = NULL;
                                        /*release the physical processor*/
 }
}
```

It may happen that the *forall* children's code is very short. By the time the parent executes instruction *EndParallel*, all the children have terminated. In this case, the parent simply proceeds to the instruction following *EndParallel*.

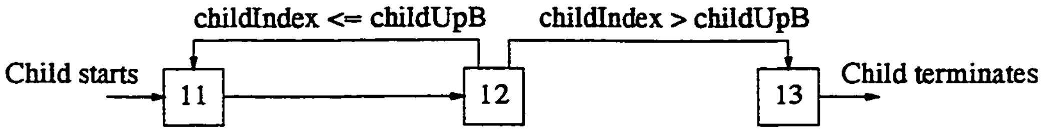
Figure 30a summarizes the parent's control flow for executing a *forall* loop. The numbers in the figure are the step numbers as denoted in Figure 25.

Child's Operations

If the new child's code does not require any parameter passing from the parent (Example *c* in Figure 23), the child starts running its code, and executes in parallel with



a) Parent's flow



b) Child's flow

Figure 30: Control flow for executing a *forall* loop

the parent. However, if the parent has parameters to send to the child (Example *a* and *b* in Figure 23), it will let the child evaluate the parameters on its behalf and go to sleep. When the child completes parameter evaluation, it will wake up the parent, and start executing its own code. The implementation of parameter evaluation upon process creation was explained in details in section 4.4.2.

The new child's code includes the body of the *forall* statement. Unlike a *fork* child that executes the body of the *fork* statement only once, a *forall* child may have to run the body of the *forall* statement more than once. This happens when *grouping* > 1. In fact, the child's code can be considered as a *for* loop whose body is the body of the *forall* statement, and the lower bound and upper bound are the child lower bound (*childLoB*) and child upper bound (*childUpB*) respectively. The index of this *for* loop is called *child forall index* (*childIndx*). *childIndx* is initialized to *childLoB* when the child is created, and incremented after each execution of the body of the *forall* statement. Thus $childLoB \leq childIndx \leq childUpB + 1$ (when $childIndx = childUpB + 1$, the child exits its *for* loop to terminate). The effect of the *for* loop is implemented by instruction *TstGrpIncIdx* (Test Grouping and Increment child's Index) (Step 12, Figure 25).

After one execution of the body of the *forall* statement (Step 11, Figure 25), the child executes instruction *TstGrpIncIdx* (Step 12, Figure 25) to test the child *forall* index. If $childIndx \leq childUpB$, the process runs the following steps:

- *childIndx* is incremented for the next execution of the child's *for* loop.
- The child's program counter is set to the starting code address of the body of the *forall* statement (Step 11, Figure 25) so that the child will run the code once more.
- Field *repeatProcInGroup* of the child's PCB is set to TRUE. The use of field *repeatProcInGroup* will be described shortly.

Therefore, if $childIndx \leq childUpB$, the child will jump back to Step 11. Steps 11 and 12 effectively implement the child's *for* loop.

If $childIndx > childUpB$, the child exits the *for* loop and proceeds to the next instruction which is *ForallChildEnd* (Step 14, Figure 25). The child will execute instruction *ForallChildEnd* to terminate. Figure 30b illustrates the child's control flow.

Field *repeatProcInGroup*

If the body of the *forall* statement is a function call with parameters (Example *a* and *b* in Figure 23), the function call is executed as shown in Figure 19. Therefore the child's code is as follows.

```
/* Step 11 */
  NewFrame;         /*allocate a function frame*/
  WakeupProcess;    /*wake up parent after parameter evaluation*/
  Call;             /*call the function*/
/* Step 12 */
  TstGrpIncIdx;     /*iterate if grouping > 1*/
/* Step 13 */
  ForallChildEnd; /*to terminate*/
```

Before the function is called, the parent is put to sleep and the child evaluates the parameter on behalf of the parent. When parameter evaluation is completed, the child wakes up the parent by executing instruction *WakeupProcess*. The waking up action should be done only once, namely in the first iteration of the child's *for* loop when the new child was just created (i.e. when *childIndex* = *childLoB*). If *grouping* > 1, the child will jump back to Step 11. However this time, the child does not have to wake up the parent since there is no parameter passing (all required parameters were passed when the child was spawned). Therefore when *childIndex* > *childLoB*, the child does not wake up the parent.

Field *repeatProcInGroup* is used to tell a *forall* child when to wake up its parent and when not to. This field is

- initialized to FALSE by instruction *NewForallChild*;
- set to TRUE by instruction *TstGrpIncIdx*;
- tested by instruction *WakeupProcess* (only if there are parameters to be passed from the parent to the new child). If *repeatProcInGroup* is FALSE, this means that the parent just created a child and was put to sleep so that the child would evaluate the parameters. The child has finished parameter evaluation, and thus wakes up the parent (Figure 20).

  If *repeatProcInGroup* is TRUE, this means that instruction *TstGrpIncIdx* was executed at least once. Thus this is not the first iteration of child's *for* loop, and the child does not have to wake up the parent.

4.5.4 *forall* Process Termination

After a *forall* child finishes its code (i.e. when *childLoB* > *childUpB*), it executes vCode instruction *ForallChildEnd* to terminate (Step 13, Figure 25). Instruction *ForallChildEnd* is the same as instruction *ForkChildEnd* (section 4.4.3), except that the function used to update the parent's state and child information is *forallDeathProcessing*() and not *forkDeathProcessing*(). Let p be the terminating *forall* child and q be p's parent. The algorithm of function *forallDeathProcessing*() is described below and summarized in Figure 31.

- Field *numForallChildren* of q's PCB is decremented to account for p's termination.
- Field *maxForallTermiTime* of q's PCB is updated (if required).

```
forallDeathProcessing()
{
  /* p: pointer to PCB of terminating process */
  /* q: pointer to PCB of the parent of the terminating child p */
  /* arrivalTime: arrival time of p's death message */

  q->numForallChildren--;
  if (q->maxForallTermiTime < arrivalTime)
    q->maxForallTermiTime = arrivalTime;
  if (q->numForallChildren == 0)  /*all q's forall children finished*/
  { q->state = Delayed;   /*unblock q*/
    q->wakeTime = q->maxForallTermiTime;
  }
}
```

Figure 31: Function *forallDeathProcessing*()

- If p is the last *forall* child that terminates, q is unblocked and its state is set to *Delayed* (q was blocked waiting for all of its *forall* children to terminate).

4.6 Channels

Write/read operations on channel variables in a CPC program abstract message send/receive in the real multicomputer. A channel can be considered as an infinite buffer owned by some process p, where other processes can deposit messages of the same type for p to read. In this section we discuss how channel buffers are implemented and how messages are deposited to and read from buffers.

4.6.1 Channel Variables

To send a message of type *msgType* to the receiver process r, the sender s identifies a channel variable v of type *msgType* owned by process r. Process s executes an assignment statement where v is on the left-hand side (LHS) of the assignment. The message content is the value of the expression on the right-hand side (RHS) of the assignment.

Process r must be aware of to which channel s has written the message (since r may have more than one channel of type *msgType*). Process r then executes a channel read on that channel to get the message from s. If the message has not arrived at r's processor, r's execution is suspended until the message is available, at which time r removes the message from the channel buffer. Examples of channel variables and operations are given in Figure 32.

Message types and thus channel types can be either basic types (integer, char, float, pointer or enumerate) or composite types (structure, array). For example, we can have a channel of *float* where every message written to or read from this channel is a *float* number. Similarly, every message written to or read from a channel of array must be an array (of a pre-determined type). Note that we cannot access individual elements of a channel of array (or structure). An entire array (or structure) must be written to or read from the channel. Examples are shown in Figure 32.

More details about channel variables and channel operations are provided in Appendix A.

```
typedef int arrayType[10];
typedef struct {
   float x, y, z;
} structType;
channel int  CI;        /*channel of integer*/
channel float CF;       /*channel of float*/
channel arrayType CA;   /*channel of array*/
channel structType CS;  /*channel of structure*/
void writer()
{
  int i, j;
  float f;
  arrayType a;
  structType s;
   ...
  /*Input i, j, f, array a, and structure s*/
  CI = (i + j)*2;  /*write to channel of integer*/
  CF = f/2;        /*write to channel of float*/
  a[5] = 0;
  CA = a;          /*write to channel of array*/
  s.y = 0.5;
  CS = s;          /*write to channel of structure*/
}
void reader()
{
  int m, n;
  float u, v, w;
  arrayType b;
  structType t;
   ...
  /*Input m, n, v and w*/
  m = (m + CI) * n;  /*read from channel of integer*/
  u = v * w * CF;    /*read from channel of float*/
  b = CA;            /*read from channel of array*/
  n = b[5];          /* n = 0 */
  t = CS;            /*read from channel of structure*/
  w = t.y;           /*w = 0.5*/
}
```

Figure 32: Examples of channel variables and operations

4.6.2 Channel Design Issues

Channel Buffers

- Write and read operations on channels abstract message send and receive respectively. When messages arrive at destinations, their contents are to be stored in buffers. In the CPSS, message buffers are also called channel buffers. The design issue is how channel buffers are implemented.

- The buffer of a channel is assumed to be infinite. That is, a channel is assumed to be able to accept an unlimited number of messages.

- Each channel is associated with one and only one channel variable and one channel ID number. The user declares and uses the channel variable in the CPC program to perform message send/receive between the process owning the channel variable and other processes. The variable owns a word in the memory pool as other basic-type variables. The CPSS uses the equivalent channel ID to execute message send/receive specified by the user.

- Messages of a channel must be read in the order they arrive at the channel. Therefore each message is associated with a timestamp which denotes the *available time* of the message.

The *available time* of a message is defined as follows. Assuming that a message arrives at the destination at time t_m, the overhead of copying the message from the router buffer to the channel buffer is b, and there are no other messages to be copied to the same channel, then the available time t_a of the message is $t_a = t_m + b$.

If several messages of the same channel arrive at the same time, then blocking time at the channel buffer is added to the actual available time of a blocked message. The blocking time is incurred by the overhead of copying messages from the router buffer to the channel buffer.

The available time can be roughly defined as the earliest time at which the message will be available for reading. The definition implies that a message needs to be written to the reader's buffer before it is actually available to the reader. The reason is that we do not route actual contents of messages. When a message is sent, the message content is copied into the reader's buffer right away. However, the value is not made available until the message actually arrives at the destination. The availability of a message in a channel buffer is determined by its available time.

Channel Operations

- Channel write/read are represented by assignment statements of C language. If the left-hand side (LHS) of the assignment is a channel variable, this is a channel write. If a channel variable exists in the expression on the right-hand side (RHS), the assignment involves a channel read. The RHS may have more than one channel variables. If both sides of the assignment statement contains channel variables, the channel read operation(s) are performed first and the final value of the RHS expression is written into the channel on the LHS.
- Each channel can have only one reader process, which is the owner of the channel.
- A channel can have many writers. However, we do not distinguish which value was written by which process.
- Message sends are non-blocking. After a process executes a channel write, it can proceed immediately to the next instruction.
- Message receives are blocking. If the reader executes a channel read and no data is available, the reader is blocked until a message arrives at that channel.
- Although the reader does not identify the writer of a message, it considers arrival times of messages. Messages must be read in the order they arrive at the channel.
- After a message is read, it is removed from the channel buffer. That is, a message can be read only once.
- A channel is said to be *opened* when it is accessed for the first time (either by a read or write). Before that the channel was *inactive*.

```
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 33: Channel data structures in C

4.6.3 Channel Descriptors

Control information of channel is stored in a structure called *channel descriptor*. When a channel is opened (i.e. accessed for the first time), it is allocated a channel descriptor which contains the following fields:

- *head*: a channel is considered to be an infinite buffer of messages of the same type. Messages written to a channel are sorted in the increasing order of writing times, and maintained in a list for reading by the channel owner. This list will be referred to as the *list of channel values*. *head* points to the first element of this list.

- *dataCount*: the current number of elements (messages) in the channel list of values

- *waitProcQueue*: when the reader is blocked on a channel read because no data is available, this field is set to point to the PCB of the reader process. When the message has arrived, the waiting reader is unblocked.

- *chanElemSize*: size of messages written to this channel

The data structure in C is shown in Figure 33.

Channel descriptors are taken from an array of channel descriptors, array `chann[]` (Figure 33), to allocate to channels when they are opened. The array of channel descriptors is a fixed-size array, and channel descriptors are allocated consecutively starting from `chann[1]`.

A channel variable is a program variable. Thus it has an entry e in the memory pool (array `storageValue[]`, Figure 6) as other variables. The index of this entry in the memory pool is called the *address* of the channel variable. The issue is what should be stored in entry e because, 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. Every subsequent access to the channel will use first the channel variable address and then the channel ID.

The steps involved in a channel access can be summarized as follows. From the channel address, the process (reader or writer) accesses the entry of the channel variable in the memory pool and obtains the channel ID. Assuming that the channel has been opened, from the channel ID, the process retrieves the channel descriptor from the array of channel descriptors. Field *head* in the channel descriptor will then point the process to the list of channel values (for reading or writing). The steps of reading a channel variable is illustrated in Figure 34.

4.6.4 Channel Buffers for Basic Types

Messages of basic types (e.g. integer, char, enumerate, float) are stored in a buffer called the *channel buffer* waiting for being read. The channel buffer is a fixed-size array and shared by messages of all opened basic-type channels. Each entry of the array buffers a message until it is consumed.

Since the channel buffer array is used for float numbers as well, every entry is associated with a tag indicating whether to interpret the value as an integer or as a float (char, enumerate and pointers are treated as integers). Figure 35 shows the data structure of the channel buffer in C. The channel buffer is array `channValue[]`.

Messages of a channel are read in the order they were written to the channel. Therefore the available time of a message must be recorded with the data itself.

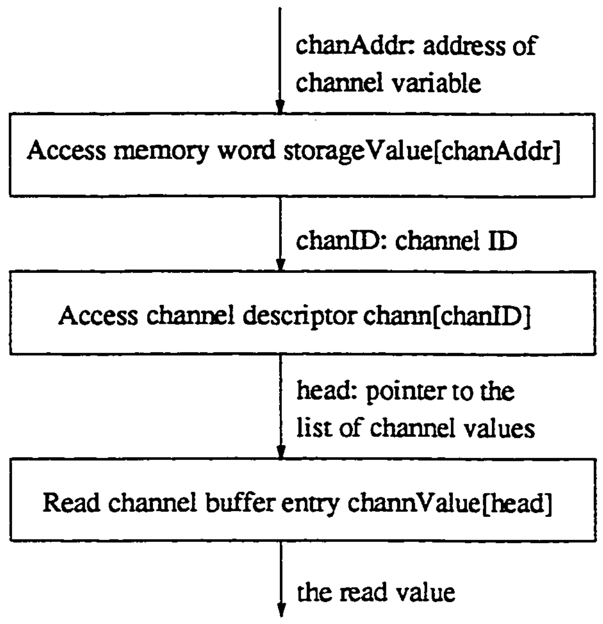

chanAddr: address of channel variable
Access memory word storageValue[chanAddr]
chanID: channel ID
Access channel descriptor chann[chanID]
head: pointer to the list of channel values
Read channel buffer entry channValue[head]
the read value

Figure 34: The steps of reading a channel

```
typedef struct { /*basic data entry for both memory pool and channel*/
   char type;    /*tag 0: int; 1: float*/
   union {
        int intValue;
        float floatValue;
   } val;
} basicValue;

basicValue channValue[MAX_NUM_CHAN_VALUES+1]; /*channel buffer*/
float channValTime[MAX_NUM_CHAN_VALUES+1]; /*message available times*/
int channValLink[MAX_NUM_CHAN_VALUES+1];
          /*to link channel values and to link free entries*/
```

Figure 35: Data structures of the channel buffer in C

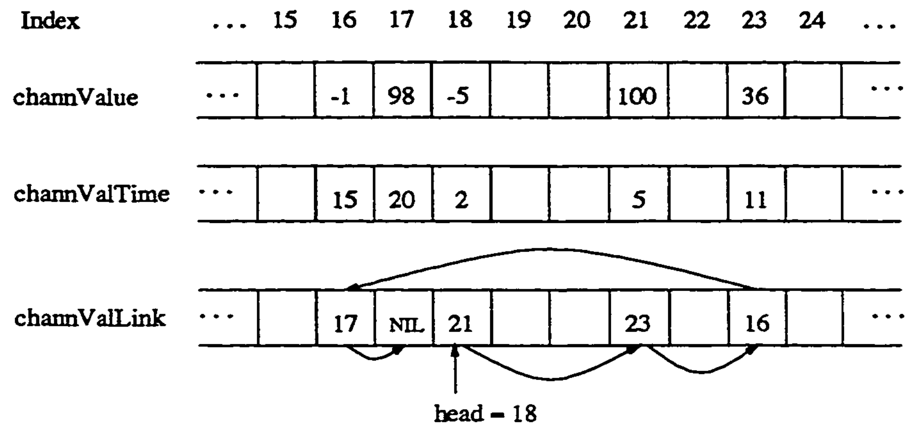


Figure 36: Implementation of the channel buffer

Another array parallel to array `channValue[]` is used to store available times of messages. This array is named `channValTime[]` (Figure 35). An entry `channValTime[i]` (`i > 0`) contains the available time of the message whose content is stored in entry `channValue[i]`. Figure 36 illustrates the two parallel arrays `channValue[]` and `channValTime[]`.

When a value (message) is written to a channel, the writer requests a free entry from the channel buffer, writes the value to the entry and inserts the entry into the list of channel values. The issue is how a list of channel values is implemented.

Messages belonging to a channel may not be stored consecutively in the channel buffer (array `channValue[]`) because the buffer is shared by all opened channels. To link the messages of a channel together into a list, we introduce another array parallel to array `channValue[]`. This array is called `channValLink[]`. Let entry `channValue[j]` follow entry `channValue[i]` in a list of channel values of a channel c (`channValue[i]` and `channValue[j]` may not be consecutive in the channel buffer). The content of entry `channValLink[i]` (corresponding to `channValue[i]`) is `j`, which is the index in the channel buffer array of the entry following `channValue[i]`. `channValLink[j]` in turn points to the entry following `channValue[j]` in c's list of channel values.

The index of the first entry of c's list of channel values is recorded in field *head* of c's channel descriptor. Figure 36 shows an example of a list of channel values pointed to by pointer *head*. The list has 5 values whose indexes in the channel buffer are 18, 21, 23, 16, and 17. This order is also the order of their available times as maintained

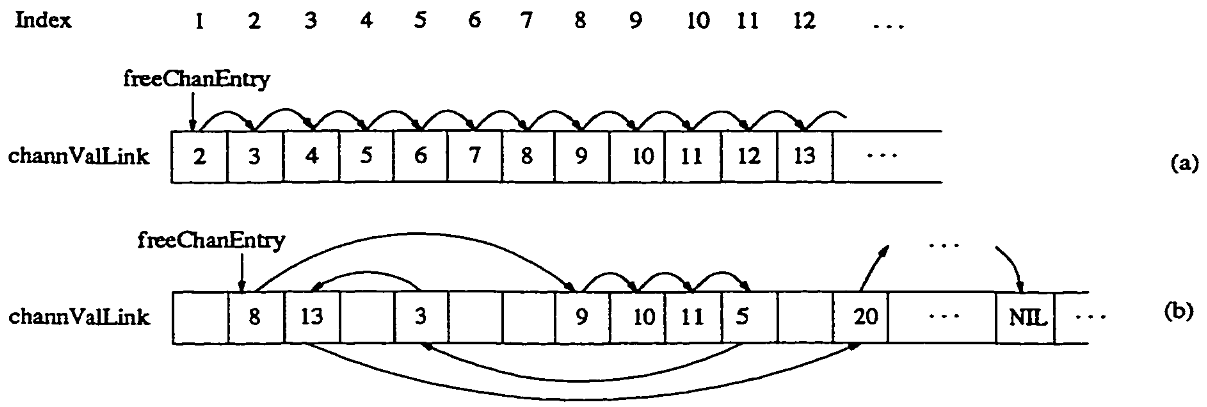


Figure 37: The list of free buffer entries (*freeList*)

by the list of values.

After a message has been read from a channel, it is removed from the list of channel values. The entry should be re-used for future messages. The freed entry is inserted into the *list of free buffer entries* (also called *freeList*). All the free entries of the channel buffer are kept on this list. When a writer process requests buffer for a new message, the front entry of the *freeList* is granted. When an entry is returned after a read, it is added to the front of the *freeList*.

The list of free entries is implemented similarly to a list of channel values. The index of front entry is recorded in variable *freeChanEntry*. An entry `channValLink[k]` (`k > 0`) corresponding to free entry `channValue[k]` stores the index of the free entry following `channValue[k]` in the *freeList*. Figure 37a depicts the *freeList* when program execution starts. All entries of the channel buffer belong to the *freeList*. The head of the list is pointed to by *freeChanEntry*. Figure 37b shows the *freeList* after several write and read operations on different channels.

Note that every entry of the channel buffer belongs to either an opened channel or the *freeList*.

In summary, channel storage and operations are implemented based on three parallel arrays:

1. `channValue[]` where contents of messages are buffered waiting to be read.

2. `channValTime[]` which records available times of messages. Message write times permit readers to know when a message will be available for reading.

The list of values of a channel is ordered based on message available times as well.

3. chann`ValLink[]` which provides links to form lists of channel values and the list of free buffer entries.

4.6.5 Channel Buffers for Composite Types

We first consider channels of structure. The same implementation will be applied to channels of array as well.

Channel of Structure

A channel of structure is a list of messages, each of which is a structure.

Consider a normal structure variable. The content of the structure is stored in a memory block, and the structure variable records the starting address of the memory block. Therefore, a channel of structure can be implemented as a list of values where each value is the starting address of a structure memory block. The structure addresses can be treated as integers and stored in the channel buffer (array `channValue[]`). Actual contents of the structures (messages) can be saved in the memory pool, one memory block for each structure. The CPSS uses this scheme to implement buffers for channels of structure.

When a message is written to a channel of structure, a memory block from the memory pool (array `storageValue[]`) is allocated. The content of the message is copied into the memory block. The starting address of the memory block (which is treated as an integer) is recorded in the channel buffer (array `channValue[]`). That is, the front entry of the *freeList* is allocated and written with the starting address of the memory block. When the message arrives at the destination, its available time is updated and the entry is inserted into the list of channel values as if this were an integer number.

In short, the list of values of a channel of structure is in fact a list of addresses, each address pointing to a structure in the memory pool. Figure 38 illustrates the storage implementation of a channel of structure; the size of each structure (message) is three words. The addresses are treated as integers, and read/write operations on the channel of structure are also based on the three arrays `channValue[]`, `channValTime[]`

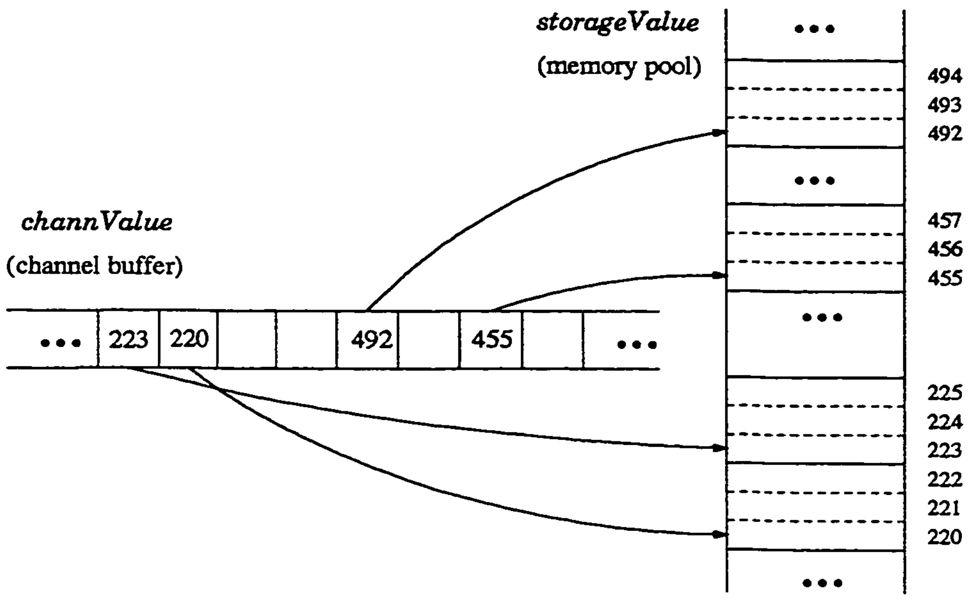


Figure 38: Implementation of composite-type channels

and `channValLink[]`.

When a process reads from a channel of structure, the first entry of the list of channel values is retrieved. From that entry, the reader obtains the starting address of the structure block in the memory pool. The reader will read the content of the structure from the memory pool. After the read is completed, the structure memory block is returned to the memory pool. The read entry in the channel buffer, which stores the structure address, is also given back to the *freeList*.

Channel of Array

An array can be considered as a structure whose elements are of the same type. Therefore channels of array are implemented in the same manner as channels of structure. That is, an entry in the channel buffer records the starting address of a memory block in the memory pool. That memory block stores the actual content of the message of type array. In summary, for a channel of array, the list of channel values is a list of memory addresses, each address pointing to an array residing in the memory pool.

4.6.6 Channel Operations on Basic Types

In this section, we consider only channel variables of basic types (e.g. integer, float, char).

Channel Write

A channel write takes the form of an assignment statement whose left hand side (LHS) is a channel variable. The message content is the value of the expression on the right hand side (RHS) of the assignment. Figure 32 shows some examples of channel writes.

A channel write consists of three steps:

1. The address of the channel variable on the LHS is loaded onto the top of the writer's stack.

   The writer will access the memory location at the indicated address, retrieve the channel ID, and use it to access the channel descriptor for the pointer to the list of channel values (Figure 34).

2. The expression on the RHS is evaluated. The final value of the expression is pushed onto the top of the writer's stack. The expression value will be the message content.

3. The writer executes vCode instruction *STChannel* (STore Channel) which writes the message content to an allocated entry of the channel buffer (array `chanValue[]`). If this is an intra-processor message (i.e. the source and destination of the message are the same physical node), the writer inserts the message into the corresponding list of channel values. If this is an inter-processor message, the writer injects the message into the network for routing. When the message arrives at the destination, the network manager will insert the message into the list of values for reading.

Steps 1 and 2 are in fact preparations for execution of *STChannel* instruction. The algorithm of *STChannel* instruction is described below and summarized in Figure 39.

- The writer restores the address of the channel variable from the top of its stack, and saves the address in a temporary variable *chanAddr*. The address of the channel variable is then popped off the stack since it is no longer needed.

- The writer accesses the memory pool at location *chanAddr* to retrieve the channel ID (*chanID*).

- If the channel has not been opened ($chanID = 0$), the writer requests a free channel descriptor from the array of channel descriptors (`chann[]`), and initializes the new channel descriptor. The new channel ID is saved at memory location *chanAddr*.

- The writer requests a free buffer entry (from the *freeList*) whose index is *bufIndex*.

- The value of the RHS expression on the top of the writer's stack is copied into the allocated buffer entry (`channValue[bufIndex]`). The read value is then popped off the stack. The writer also updates the list of free buffer entries (pointer *freeChanEntry*).

- If this is an intra-processor message, the actual available time of the message is computed, which also takes into account buffer write contentions if any. The writer then inserts the new buffer entry into the list of values of the written channel by executing function *insertValueInChanQ*(). This function will be described below.

  If the new message is inter-processor, the writer sets the message available time to a very big value which indicates that the message has not arrived at the destination yet, and the arrival time is unknown for the time being. The writer process then injects the message into the network for routing by executing function *WH_CEM_SENDS_MSG*(). The network manager receives the following information from the writer: source and destination node IDs, message length, send time, channel ID of the written channel, and index of the buffer entry *bufIndex*. When the message reaches the destination, the network manager will calculate the actual available time of the message, and insert it into the list of values of the written channel by calling function *insertValueInChanQ*() as well.

In function *WH_CEM_SENDS_MSG*(), a *message* structure is allocated and intialized with the message information passed by the writer process. The *message* is then inserted into the list of new messages, waiting for being routed.

```
STChannel
{
 /*Pre-conditions: the address of the channel variable and
     the value of the RHS expression are on the stack top*/

 chanAddr = address of the channel variable; /*obtained from stack top*/
 Pop address of channel variable off the stack after using it;
 chanID = storageValue[chanAddr];

 if (chanID == 0) /*channel has not been opened; get a new channel descriptor*/
 { chanID = index of the allocated channel descriptor in array chann[];
   storageValue[chanAddr].val = chanID;
   Initialize the new channel descriptor;
 }
 /*Request an entry from the channel buffer*/
 bufIndex = index of the allocated entry in array channValue[];
 Initialize the new buffer entry {
   channValue[bufIndex] = value on the stack top; /*this is the value of
       the RHS expression pushed in before (Step 2 of a channel write)*/
   channValLink[bufIndex] = NULL; }
 Pop the top value off the stack; /*it has just been used*/
 Update the list of free buffer entries; /*update pointer freeChanEntry*/

 if (physical destination == physical source) /*intra-processor message*/
 { Compute message available time;
             /*considering buffer write contention, if any*/
   channValTime[bufIndex] = available time;
   InsertValueInChanQ(chanID, bufIndex, availableTime); /*insert entry
       bufIndex into list of channel values using computed available time*/
 }
 else /*inter-processor message*/
 {  channValTime[bufIndex] = infinity; /*arrival time not known yet*/
   WH_CEM_SENDS_MSG(source, dest, msgSize, sendTime, chanID, bufIndex);
       /*inject the new message into the network for routing*/
 }
} /*end STChannel*/
```

Figure 39: Algorithm of vCode instruction *STChannel*

The algorithm of function *insertValueInChanQ*() is as follows.

- The function considers the available time of the new message, and inserts the message into the list of channel values. The insertion position respects the increasing order of available times of the messages in the list.
- If the owner of the channel is currently blocked due to a previous channel read, the owner process is unblocked: its state is changed to *Delayed* and the wakeup time is the available time of the new message.
- Field *dataCount* of the channel descriptor is incremented to account for the new message.

After completing the channel write, the writer process proceeds to the next instruction of its code. As implemented, channel writes are non-blocking.

Channel Read

A channel read is performed when a channel variable is present on the RHS of an assignment statement. In Figure 32, the process executing function *reader*() performs a read on channel *ch*. If no data is currently available, the reader process will be blocked and queued at the channel queue (field *waitProcQueue* of the channel descriptor); we say that the channel read is *suspended*. When a message is available for reading, the reader will resume execution and re-run the channel read to get the value.

If the data is available, the content of the first entry of the list of channel values is returned, and the entry is removed from the list. In this case, the channel read is said to be *successful*. After a channel read is completed successfully, the read value is treated as if it belonged to a normal variable. That is, the value read from the channel will be pushed on the top of the reader's stack and used for evaluating the RHS expression. Therefore the post-condition of a successful channel read is that the top of the reader's stack contains the read value.

A channel read consists of two steps:

1. The address of the channel variable to be read is calculated and stored in a temporary variable *chanAddr*.

2. The reader executes vCode instruction *LDChannel* using the address of the channel variable stored in *chanAddr*.

The pre-condition of *LDChannel* is that the address of the channel variable to be read is stored in a temporary variable *chanAddr*. The algorithm of instruction *LDChannel* is as follows.

- The reader verifies if it is really the owner of the channel to be read.
- If so, the reader accesses the memory pool at location *chanAddr* to obtain the channel ID (*chanID*).
- If the channel has not been opened ($chanID = 0$), the reader requests a free channel descriptor from the array of channel descriptor (`chann[]`), and initializes the new channel descriptor. The new channel ID is saved at memory location *chanAddr*.
- The reader's channel read status (field *readChannStatus* of the PCB) is set to *AtChannel* (i.e. in the middle of a channel read).
- If the channel is empty, the reader's state is set to *Blocked*. If the channel is not empty but the data is not available yet (i.e. the reader's local clock has not reached the available time of the message), the reader's state is changed to *Delayed* and the wakeup time is set to the available time of the message. In either case, the following steps are also executed.
  - The reader releases the processor on which it is running since it will be blocked until the data is available.
  - The reader is queued at the channel queue: field *waitProcQueue* of the channel descriptor is set to point to the reader's PCB.
  - The reader's program counter is decremented by one so that when the message becomes available later, the reader will re-run this instruction (*LDChannel*) to read the channel again.
  - The reader exits from instruction *LDChannel* and goes to sleep.

  If the data is currently available, the reader gets the message from the channel buffer and updates channel information as follows.

- The content of the first entry in the list of channel values is copied on top of the reader's stack.
- The read entry is removed from the list of channel values, and returned to the list of free entries.
- The channel descriptor is updated: field *head* is set to point to the next entry of the list of channel values; field *dataCount* is decremented for one less message; the pointer to the reader's PCB at the channel queue (field *waitProcQueue*), if any, is removed.

The post-condition of *LDChannel* is that if the channel read is successful, the read value is pushed on top of the reader's stack. The algorithm of instruction *LDChannel* is summarized in Figure 40.

4.6.7 Channel Operations on Composite Types

In the following discussion, we use channels of array as examples. Read and write operations on channels of structure are performed in the same manner.

Channel Write

As described earlier, the list of values of a channel of array is a list of memory addresses, each address pointing to a memory block in the memory pool which stores the actual content of the array message.

We also mentioned that it is not allowed to read/write an individual element of a message of type array. The whole array must be copied into a normal array variable which then enables access to each individual array element. In Figure 32, the assignment statement $CA = a$ is a write on channel of array CA. Array a is a normal array.

Write operation to a channel of array is indeed similar to write to a channel of integer. The difference is the extra work required to copy the contents of the array on the RHS of the assignment statement to a new memory block requested from the memory pool for the array message. Following is the algorithm for a write to a channel of array, which consists of four steps.

1. The address of the channel of array variable (on the LHS of the assignment statement) is pushed onto the top of the writer's stack.

```
LDChannel
{
 /*Check if this process is the owner of the channel*/
 chanOwner = storageOwner[chanAddr];
 if (chanOwner != reader's processor ID)
  { set system state to ChanReadErr (Channel Read Error); break; /*exit*/ }
 chanID = storageValue[chanAddr];
 if (chanID == 0) /*channel has not been opened; get a new channel descriptor*/
 { chanID = index of the allocated channel descriptor in array chann[];
   storageValue[chanAddr].val = chanID;
   Initialize the new channel descriptor;
 }
 chanPtr = &chann[chanID]; /*pointer to the channel descriptor*/
 readerPtr = pointer to the PCB of the reader process;
 if (readerPtr->readChannStatus == None)
   readerPtr->readChannStatus = AtChannel; /*in middle of a channel read*/
 dataIndex = chanPtr->head; /*first entry of list of channel values*/
 if ((chanPtr->dataCount == 0)  /*channel is empty*/
  or (channValTime[dataIndex] > readerPtr->time)) /*data not yet available*/
 {  /*the reader process is put to sleep*/
   release the reader's processor; /*set runProc to NULL*/
   channPtr->waitProcQueue = pointer to the reader's PCB;
   if (chanPtr->dataCount == 0)  /*channel is empty*/
     readerPtr->state = Blocked;
   else  /*channel is not empty, but data is not available yet*/
   { readerPtr->state = Delayed;
     readerPtr->wakeTime = channValTime[dataIndex]; /*available time*/
   }    readerPtr->PC--; /*next time this channel read will be re-executed*/
 }
 else  /*data is available for reading*/
 { copy value channValue[dataIndex] onto top of reader's stack;
   return the read entry to the freeList;
   chanPtr->head = channValLink[dataIndex]; /*next entry of list of values*/
   chanPtr->dataCount--; /*one less entry in list of values*/
   if (channPtr->waitProcQueue) /*waiting queue is not empty*/
     remove the waiting entry from the queue;
   readerPtr->readChannStatus = None; /*successful read*/
 }
} /*end LDChannel*/
```

Figure 40: Algorithm of vCode instruction *LDChannel*

2. The starting address of the normal array (on the RHS of the assignment statement) is pushed onto the top of the writer's stack. Steps 1 and 2 are to prepare for instruction *CopyToNewBlock* in step 3.

3. The writer executes vCode instruction *CopyToNewBlock* which does the following tasks:

   - get a block of the size of the array from the memory pool;
   - copy the contents of the array on the RHS to that newly allocated block;
   - pop the address of the RHS array variable off the stack (the address was pushed onto the stack in step 2);
   - push the starting address of the new block onto the top of the writer's stack, preparing for instruction *STChannel* in step 4.

   After step 3, the writer's stack top contains the address of the LHS channel variable and the value to be written into the channel buffer. This value is the starting address of the memory block that stores the actual contents of the message of type array.

4. The writer executes instruction *STChannel* which treats the starting address of the memory block containing the message content as an integer. This integer is inserted into the list of values of the channel of array. The algorithm of instruction *STChannel* was presented in section 4.6.6.

When the array message arrives at the destination node, the same code for channels of integers is executed for channels of array. That is, the network manager first computes the actual available time of the message. The network manager then calls function *insertValueInChanQ*() to insert the address of the new array into the list of channel values.

Channel Read

An example is the assignment $b = CA$ in Figure 32, where b is a normal array and CA is a channel of array. A read operation on a channel of array consists of three steps.

1. The starting address of the normal array (on the LHS of the assignment statement) is pushed onto the top of the reader's stack.

2. The reader executes instruction *LDChannel* which accesses the list of channel values and copies the first entry onto the top of the reader's stack (if the data is available for reading). The first entry contains the starting address of the array (message) to be read. The algorithm of instruction *LDChannel* was presented in section 4.6.6.

   After step 2, the reader's stack top holds the starting addresses of the LHS array and the array (message) to be read. This is the pre-condition for vCode instruction *CopyBlock* executed in step 3.

3. The reader executes instruction *CopyBlock* which copies the content of the array message to the LHS array. *CopyBlock* also pops the top two values off the reader's tack.

Chapter 5

The Wormhole-Routed Network Simulator

This chapter describes the design and implementation of the wormhole-routed network simulator. We first discuss candidate network parameters needed for simulation accuracy. Major data structures of the network simulator and their operations are then described. We also present simulator algorithms which include communication step for advancing unblocked flits, and approximate round-robin scheduling algorithm for link bandwidth allocation.

5.1 Network Parameters

Communication model was described in Chapter 3. In this section we present candidate network parameters which are essential for accurate network simulation.

5.1.1 Network Topology

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

Low-dimensional topologies (such as line, ring, mesh and torus) offer higher wire efficiency. However, the average distance between nodes is relatively large.

For networks in which the routing latency depends on the path length (such as packet switching), hypercube is the most popular choice because of its short internode distance. However, in networks supporting circuit switching or wormhole routing, the network latency is almost independent of the path length if the contention is absent and the packet length is relatively large. In this case, low-dimensional meshes and tori are favorite topologies.

In our simulator, we support both low and high dimensional topologies. Specifically, the following topologies are simulated: line, ring, 2D-mesh, 2D-torus, 3D-mesh, 3D-torus and hypercube.

5.1.2 Network Size

From the simulator design point of view, the maximum network size that can be simulated depends on the flit size and the buffer size of virtual channels. If each virtual channel has a single-flit buffer and the flit size is k bits, the largest network can have up to 2^k nodes. The reason is that every packet must have a flit carrying the address of the destination node. This information must be received completely by each intermediate node on the path of a packet to compute the address of the next node on the path.

In the above case, if we wish to simulate networks of size larger than 2^k while keeping the flit size unchanged, the buffer size of virtual channels must be extended. For example, a network with a flit size of 8 bits and a single-flit buffer at each virtual channel can have at most 256 nodes. To simulate networks with more than 256 nodes and the same flit size, the buffer size of virtual channels must be at least 2 flits. In this case, intermediate nodes on the path of a packet can receive the complete address of the destination node from the buffer.

Our simulator can simulate networks of unlimited sizes (provided that the flit size and the buffer size of virtual channels are properly set). The only limitation is the memory capacity of the host computer.

5.1.3 Virtual Channels

A virtual channel consists of a buffer that can hold one or more flits of a packet and associated state information [15]. Several virtual channels may be multiplexed on a physical link and share the link bandwidth. Advantages of virtual channels include the following:

- Virtual channels enable deadlock-free routing algorithms [15] and many adaptive routing algorithms [39, 40, 41, 42].
- Virtual channels allow many packets to share the bandwidth of a link. This improves link utilization and enhances network throughput [17, 18].
- By increasing the degree of connectivity in the network, virtual channels facilitate the mapping of a virtual topology onto a different physical topology, especially when the physical topology has lower connectivity [8].
- Extra connections provided by virtual channels are useful to route packets around congested or faulty nodes [8].
- Virtual channels provide the ability to deliver guaranteed communication bandwidth to certain class of packets. This ability is important to build real-time networks. Virtual channels ensure that some minimum link bandwidth can be allocated to each virtual channel provided that the number of virtual channels sharing the same link is bounded [8, 29].

Design issues regarding virtual channels are discussed below.

Virtual Channel Abstraction

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

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

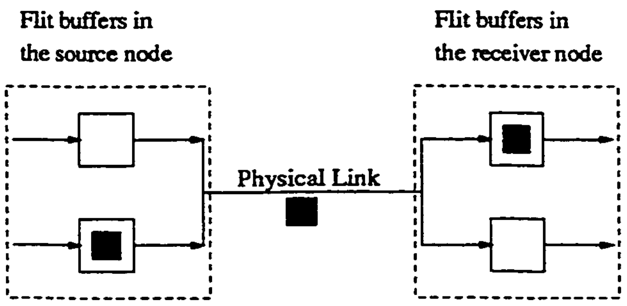

Flit buffers in the source node
Flit buffers in the receiver node
Physical Link

Figure 41: Two virtual channels sharing a physical link

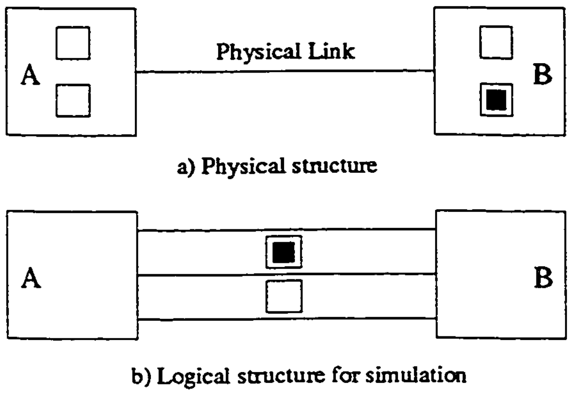


Figure 42: Virtual channel abstraction for simulation

Two virtual channels multiplexed on a physical link are shown in Figure 41. There are two flit buffers in the source node and two flit buffers in the receiver node. One source flit buffer is paired with one receiver flit buffer to form a virtual channel when the link bandwidth is allocated to the pair. In the network simulator, such a virtual channel is abstracted into an entity called *lane* [18]. Each lane is associated with a lane buffer which represents a flit buffer at the receiver node. When a flit is deposited into a lane buffer, it is considered to be stored in a flit buffer at the receiver node. This is illustrated in Figure 42.

An example of a flit transfer from one node to the next node is shown in Figure 43. Let links a and b be connected to nodes A and B respectively as illustrated in Figure 43. In reality, the flit is moved from buffer 1 of node A to buffer 2 of node B. In the simulation, the flit is transferred from the buffer of lane 1 of link a to the buffer of lane 2 of link b.

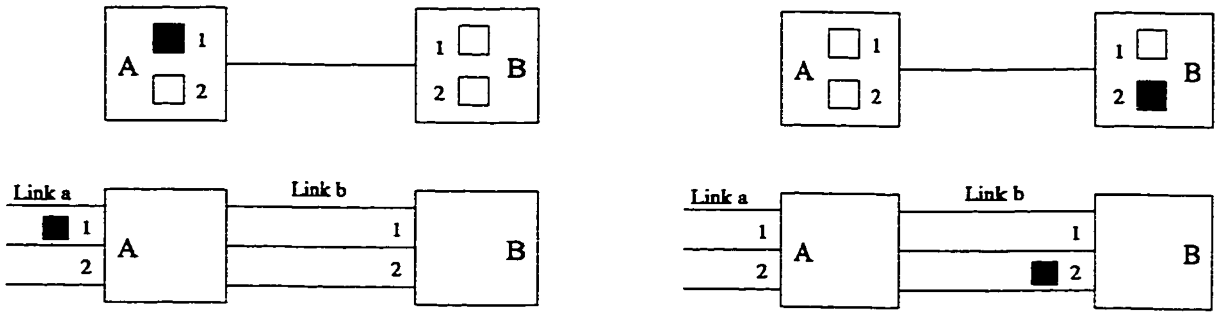


Figure 43: Simulation of a flit transfer

This abstraction does not affect simulation accuracy, and helps to speed up simulation time. The terms "virtual channel" and "lane" will be used interchangeably in this chapter.

Number of Virtual Channels

Adding more virtual channels help to improve network utilization [17, 18]. However, as the number of virtual channels increases, link scheduling becomes more complicated, requiring additional hardware complexity. In addition, time-sharing of link bandwidth may increase network latency if the traffic is high because the link bandwidth is divided among several messages.

Therefore the trade-off between enhanced network throughput and longer routing latency should be weighted when deciding the number of virtual channels per link.

In our simulator, every physical link has the same number of virtual channels, and the number of virtual channels per link is a configurable parameter. As the number of lanes increases, space requirements for simulated lanes also increase. However the increase in the number of lanes per link incurs little simulation time overhead.

Unidirectional versus Bidirectional Channels

Virtual channels can be unidirectional or bidirectional. We can have a pair of opposite unidirectional channels between two adjacent nodes. The implementation and control of this scheme are simple. However, one channel may be very busy while the other is idle; the physical link is thus not fully utilized.

Link utilization can be increased by combining two unidirectional lanes into a single bidirectional lane. If the bandwidth of each unidirectional lane is W, then the bandwidth of the corresponding bidirectional lane is $2W$. Link utilization increases at the cost of more complex implementation. For instance, a special arbitration line is needed between two adjacent nodes connected by a bidirectional channel to control the direction of information flow.

In our network simulator, we assume that bidirectional virtual channels share a bidirectional physical link.

Buffer Size

Each virtual channel is associated with a buffer. The buffer size of a virtual channel is an integral multiple of the flit size. Virtual channel buffers are FIFO queues. Larger buffer size may improve network performance [8, 29], but buffer size equal to packet size will effectively reduce the benefit of wormhole routing to that of packet switching.

Increasing buffer size helps to support large networks when the flit length is not long enough to carry node addresses (as discussed in section 5.1.2).

Our network simulator allows the buffer size to be changed whenever needed. The minimum value is 1 flit; the maximum value is the packet size in flits.

Virtual Channel Allocation Policy

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

Link Bandwidth Allocation Policy

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

To preserve link bandwidth, only those virtual channels which have a non-empty flit buffer at the sending side and a non-full flit buffer at the receiving side may participate in the scheduling decision (such virtual channels are called *active lanes*). That is, if k lanes are multiplexed on a physical link with bandwidth W, and a lanes are active ($1 \leq a \leq k$), then each active lane should get an effective bandwidth of W/a. Since the number of active lanes changes over time, the router should be able to dynamically allocate link bandwidth to the active lanes so as to maximize link utilization.

In our network simulator, round-robin scheduling is used to allocate the bandwidth of a physical link to its virtual channels.

5.1.4 Messages

A message is the logical unit of information for interprocess communication. Messages may have variable lengths. A message is often divided into a number of fixed-length packets for routing.

Our simulator supports messages of any length, and messages are packetized for routing.

5.1.5 Packets

A packet is the basic unit carrying the address of the destination node for routing purposes. Depending on network conditions, packets belonging to a message may arrive at the destination node out of sequence. Thus a sequence number is required in each packet to allow reassembly of the message.

In wormhole-routed networks, a packet is further divided into a number of fixed-length flits. There are two types of flit: control flit and data flit. The destination address and the sequence number are control flits and occupy the header flits. The remaining flits are the actual data of the packet (Figure 44).

Typical packet size ranges from 8 to 64 bytes. Factors influencing the choice of packet size include the routing scheme, link bandwidth, router design and network traffic intensity [23].

The sequence number may need one or two flits depending on the message length. The destination address may also occupy two flits if the flit length is not long enough

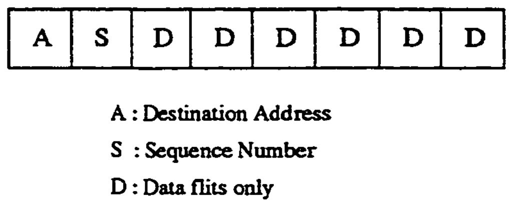


Figure 44: Structure of an 8-byte packet

to represent node addresses (in this case, the virtual channel buffer size must be at least two flits to store the complete address of the destination).

Packet size is a configurable parameter in the CPSS network simulator.

5.1.6 Flits

In wormhole-routed networks, flit is the smallest unit of information transmission.

The network size affects the flit size that must be long enough to represent node addresses. For instance, a 256-node network requires 8-bit flits. To keep the flit size unchanged and support larger networks, the size of virtual channel buffers must be extended as mentioned above.

In section 2.2.1, we see that the network latency for wormhole routing is $(F/B)D+P/B$ where F is the flit size (in bits), P the packet size (in bits), B the channel bandwidth (in bits/second), and D the path length. If $F << P$, the path length D will not affect the latency much (unless the path is very long). Thus small flit size helps to reduce network latency.

Special attention should be paid to real-time networks. Such networks usually employ many kinds of prioritized queues based on message priorities. Link bandwidth allocation is based on message priorities and performed at the flit level. If the flit size is too small, flit overheads may outweigh the benefits of wormhole routing. In real-time systems, the flit size should be somewhere between four and sixteen bytes [29].

Other factors deciding the choice of flit size include the routing scheme, link bandwidth, and router design [23]. The CPSS network simulator allows users to change the flit size as desired.

5.1.7 Message Startup Overheads

The startup cost of a message can be on the order of a thousand instruction cycles [52, 53, 54]. This is due primarily 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.

In the network simulator, message and packet startup overheads are configurable parameters.

5.1.8 Routing Scheme

Routing schemes can be *deterministic* or *adaptive*.

- Deterministic routing: The routing path of a message is determined in advance based on the source and destination addresses and independent of current network conditions. This may lead to higher latency but the implementation is simple.

- Adaptive routing: To alleviate network congestion, an adaptive scheme may dynamically re-route the message using additional information such as resource conflicts and presence of alternative paths. Adaptive routing responds well to network condition to result in higher throughput. However this is achieved at the cost of increased hardware complexity and more potential for deadlock.

Deterministic and adaptive routing can be *minimal* or *non-minimal*. Using a minimal routing algorithm, a packet is delivered through one of the shortest paths connecting the source and the destination. In a non-minimal routing algorithm, the path for delivering a packet may not be the shortest path.

Our simulator supports only deterministic routing. For line, mesh and hypercube, we employ dimension-ordered routing. Using dimension-ordered routing, each packet is routed in one dimension at a time, arriving at the proper coordinate in each dimension before switching to the next dimension. By enforcing a strictly monotonic order on the dimensions traversed, the network is guaranteed to be deadlock free. In particular, we use shortest-path routing for line topology, XY routing for 2D-mesh, XYZ routing for 3D-mesh, and E-cube routing for hypercube. These routing schemes are minimal.

For ring and torus topologies, it is impossible to construct a deadlock-free minimal deterministic routing algorithm [8]. We provide minimal deterministic routing for rings and torus, which may cause deadlock. To avoid deadlock, we also support deadlock-free deterministic routing for ring and torus; the algorithm was proposed by Dally and Seitz[15]. This deadlock-free routing algorithm is non-minimal though.

Above are default routing schemes supported by the CPSS network simulator. The CPSS also permits users to specify their own routing algorithms using a routing table.

With deterministic routing, routing decisions can be *explicit* or *implicit*

- Explicit routing: the source node computes the entire route in advance, before sending out the message. The router will simply switch accordingly.

- Implicit routing: the routing path is determined by the routers on the path. Each router will compute the next node on the path based on the addresses of the current node and the destination.

Our simulator uses an implicit scheme to shift the burden of routing to the routers which are usually dedicated hardware for communication tasks. Furthermore, support for implicit routing allows adaptive routing to be added easily to the CPSS network later.

5.1.9 Flit Routing Latency

The latency for a flit to move from one node to the next node on the path consists of the following components [26]:

- Buffer read time: the time needed to read the flit from the current buffer.

- Link scheduling time: the time required to schedule the physical link in order to know which lane is allowed to use the link in the current clock cycle.

- Link delay time: the time taken by the flit to traverse the link and arrive at the next node.

- Buffer write time: the time needed to write the incoming flit to the buffer at the next link.

In our simulator, buffer read time, link scheduling time, link delay time and buffer write time are combined into one parameter which is *flit latency.* All flits require flit latency to transfer from one node to another.

In addition to the flit latency, header flits incur router decision time and lane allocation time.

- Router decision time: the time required by the router to determine the next node on the path. This time depends on the routing algorithm and message address format.

- Lane allocation time: the time needed to allocate a free lane of the next link where the flit will be deposited. This time depends on whether there are free lanes on the next link.

Router decision time and lane allocation time are covered by the parameter called *header overhead.* Users are allowed to define the flit latency and the header overhead.

5.2 Data Structures and Their Operations

5.2.1 Network Clock

The simulated network does not have an explicit clock. Each flit transfer from one node to the next node is assumed to take one time unit (the flit latency). Other network time parameters are normalized using the flit latency unit. Note that the computation quantum is also computed based on the flit latency unit as discussed in Chapter 4.

The network uses a timestamp generator *clock_wh* to schedule physical links (i.e. link bandwidth allocation). The timestamp generator is incremented every time all unblocked flits are advanced by one link.

5.2.2 Nodes

Nodes in the network are identified by absolute IDs. Absolute IDs are computed as follows.

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

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

5.2.3 Links

Link Numbering

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

- Ring: Assume that the ring network has n nodes numbered from 0 to $n-1$. Each node in a ring topology is connected to two links: one to the left and the other to the right of the node. Because in the simulator links are assumed to be bidirectional, the total number of links is n. To make the link numbering scheme easy to understand, we consider that each node of the ring is "in charge" of the link to its right, and this link has the same ID as the node. Figure 45a shows an example with $n = 4$.
- Line: Line topology uses the same link numbering as ring. However the two boundary links connected to nodes 0 and $n-1$ are not used (Figure 45b).
- 2D-Torus: Assuming the torus has R rows and C columns, absolute node IDs then run from 0 to $R \cdot C - 1$. Each node of the torus is connected to four links to

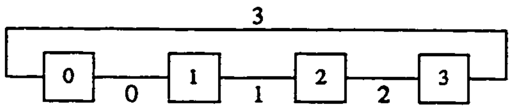
the east, west, north and south side of the node. Since links are bidirectional, the total number of links is $2R \cdot C$. We consider that each node k ($0 \leq k < R \cdot C$) of the torus is "in charge" of two links: link $2k$ to the east side and link $2k+1$ to the south side of the node. An example is given in Figure 45c.

- 2D-Mesh: The link numbering scheme of 2D-torus is also used for 2D-mesh. However boundary links are not used (Figure 45d).
- 3D-Torus: Assuming that the 3D-torus has P planes, R rows and C columns, absolute node IDs then run from 0 to $P \cdot R \cdot C - 1$. Each node of the torus is connected to six links to the east, west, north, south, front and back side of the node. The total number of links is $3P \cdot R \cdot C$ since the links are bidirectional. We consider that each node k ($0 \leq k < P \cdot R \cdot C$) of the torus is "in charge" of three links: link $3k$ to the east, link $3k+1$ to the south, and link $3k+2$ to the back side of the node.
- 3D-Mesh: The same link numbering of 3D-torus is applied to 3D-mesh. However boundary links are not used.
- Hypercube: Let i and j be the absolute IDs of two adjacent nodes. Without loss of generality, assume that $i < j$. Let d be the number of dimensions of the hypercube, and b be the position of the bit where i and j are different (with the rightmost bit being at position 0, $0 \leq b < d$). The ID of the link connecting nodes i and j is $i \cdot d + b$.

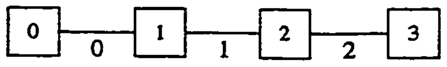
Link Data Structure

Every link is associated with a structure which stores the following information/data structures:

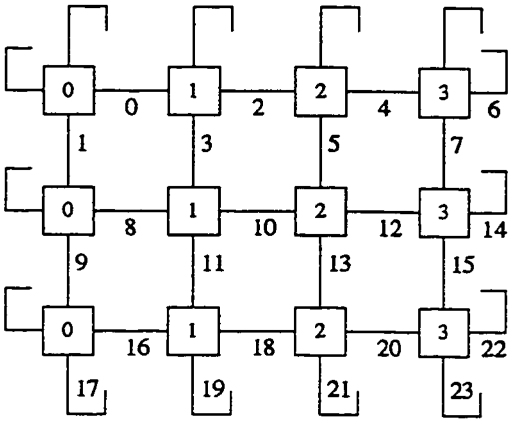
- *lane*: an array of dynamically allocated *lane* structures.
- *nbrBusyLanes*: the current number of busy lanes.
- *scheduledLane*: the lane scheduled to use the link bandwidth in the current network cycle.



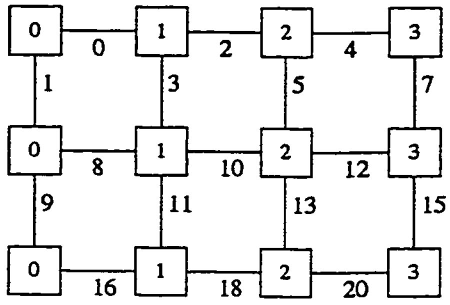
a) Ring (n = 4)



b) Line (n = 4)



c) 2D-torus (R = 3, C = 4)



d) 2D-mesh (R = 3, C = 4)

Figure 45: Link numbering

- *schedTime*: the most recent scheduling time. If this time is equal to the value of the timestamp generator, it means that the link has been scheduled in the current network cycle.

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

The structure in C is as follows.

```
struct link
{
   Lane *lane;
           /*set of lanes of the link (an array of lane structures)*/
   int nbrBusyLanes;      /*number of busy lanes*/
   int scheduledLane;     /*lane scheduled to use the physical link*/
   float schedTime;       /*the last scheduling time*/
   QueueEntryPtr queueHead, queueTail;
```

```
                          /*queue of packets waiting for free lanes*/
};
```

5.2.4 Virtual Channels

Each virtual channel (lane) has a structure which contains the following data:

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

Following is the *lane* structure in C.

```
struct lane
{
   PacketPtr  packetPtr;  /*packet currently occupying the lane*/
   enum LaneStates state; /*lane state*/
   int nbrFlits;          /*current number of flits in the lane*/
   int lastFlitPassed;
                       /*ID of the most recent flit leaving the lane*/
   int prevLink;          /*previous link on the path of the packet*/
   int prevLane;          /*previous lane on the path of the packet*/
};
```

In the simulator, virtual channels are resources. If a packet is requesting a lane from a link and no lane is available, the packet is appended to the link queue and its state is set to *Blocked.* In the current implementation, the queue at each link is maintained using FIFO order. When a lane becomes free, the first waiting packet is removed from the queue and given the lane. Other allocation schemes (e.g. random or priority queue) can be easily added to the simulator.

5.2.5 Messages

In the CPSS, there are three kinds of messages:

- Birth messages that are sent when new processes are created.
- Death messages that are sent by child processes to their parents when the children terminate.
- Channel-write messages that are generated by writes to channel variables.

Information needed to route a message is stored in a *message* structure. All messages require the following information:

- *source*: the address of the source node.
- *dest*: the address of the destination node.
- *nbrPackets*: the message length measured in terms of the number of packets.
- *sendTime*: time at which the message will be injected into the network. This time includes all message and packet startup overheads.
- *next* and *prev*: pointers to form doubly-linked lists of messages.

A birth message is destined to a new child process. Upon creation, the child process is put to sleep until the birth message arrives at the destination node on which the child will be running. At that time, the child process will be waken up to run. Therefore, the pointer to the process control block of the child is also recorded in the *message* structure.

Similarly, when a child process sends a death message in order to terminate, the pointer to the process control block of the parent process is stored in the *message* structure.

When a channel-write message arrives at the destination, the arrival time must be recorded with the written value so that the reader knows if the value has been available for reading yet. Therefore, the ID of the channel variable and the buffer location where the value is stored must also be kept in the *message* structure.

Thus depending on the message type, the *message* structure also contains the following information:

- *chanVarNum*: the ID of the channel variable,
- *chanValIdx*: the buffer where the message is deposited for reading,
- *processToNotify*: the process to be notified (if any) when the message arrives at the destination.

The *message* structure in C is as follows.

```
struct message
{
   int source;      /*source node ID*/
   int dest;        /*destination node ID*/
   int nbrPackets;  /*number of packets*/
   float send_time; /*injection time including all overheads*/
   struct message *next, *prev;
                     /*linked lists of new and active messages*/

   /*Info used to update a channel variable or a process of the CEM*/
   int chanVarNum,  /*ID of the channel variable*/
       chanValIdx;  /*message buffer location (index)*/
   ProcDesPtr processToNotify;
       /*pointer to process to be notified when the message arrives*/
}
```

For every message, we consider two kinds of startup overheads:

- Message startup overhead t_m: startup overhead for a message on initialization,
- Packet startup overhead t_p: cost to initialize a packet for routing.

For every message the total startup overhead is $t_m + p \cdot t_p$ where p is the number of packets contained in the message.

The simulator does not route actual contents of messages but simulates the routing using message information stored in *message* structures.

When a process generates a message, the CEM passes the message information to the network manager. The network manager allocates a *message* structure to store this information, and calculates the actual send time of the message which includes all startup overheads. The *message* structure is then inserted into the *list of new messages* that is ordered by actual send times of messages.

At the beginning of every communication quantum, the network manager scans the list of new messages. If the actual send time of a new message has come, the *message* structure is removed from the list of new messages and appended to the *list of active messages.* This list contains messages that are currently being routed. The network manager also cuts the message into packets and appends the *packet* structures to the *list of packets*, preparing to route the packets.

When all the packets of a message have been received by the destination node, the corresponding *message* structure is removed from the list of active messages and freed.

5.2.6 Packets

Every message is decomposed into one or more packets. Data needed for routing a packet is stored in a *packet* structure which contains the following fields:

- *msgPtr*: pointer to the *message* structure. Message information need not be replicated in the *packet* structure.
- *state*: packet state which takes one of the following values: *Init* (just initialized), *InitBlocked* (just initialized and being blocked), *Advancing* (being routed), *Blocked* (being blocked).
- *headNode*: the node where the header flit is currently buffered,

- *headLink*: the link where the header flit is currently buffered,
- *headLane*: the lane where the header flit is currently buffered,
- *next* and *prev*: pointers to form a doubly-linked list of packets.

The *packet* structure in C is as follows.

```
struct packet
{  MsgPtr msgPtr;          /*pointer to the msg structure*/
   enum PacketStates state;   /*packet state*/
   int headNode;           /*head node*/
   int headLink;           /*head link*/
   int headLane;           /*head lane*/
   struct packet *next, *prev;  /*to form the linked list of packets*/
}
```

In the network simulator, packet size is modifiable. Every packet is composed of header flits and data flits. Header flits store the destination address and the sequence number of the packet. The header size thus depends on the network size and the message length. The buffer size also affects the header size indirectly: if the header flit is not long enough to store the whole destination address (or the packet sequence number), the buffer size must be increased to accommodate more flits to store the complete destination address.

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

5.2.7 Flits

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

The current position of the header flit of a packet is recorded in the *packet* structure, fields *headLink* and *headLane* (section 5.2.6).

The position of the tail flit is not kept track of explicitly. When the tail flit of a packet leaves a lane buffer, that lane should be deallocated. To implement this, we use field *lastFlitPassed* of the *lane* structure (section 5.2.4). This field records the ID of the flit which left the lane the most recently. Let the packet size be p, and the flits of a packet be numbered from 1 to p. When field *lastFlitPassed* of a lane L reaches value p, this means that the tail flit of the packet occupying L just left the lane. L can thus be released.

5.2.8 Routing Scheme

Our simulator currently supports only deterministic routing. The routing is specified in a *routing table* that is a 2D-array. When a header flit arrives at a node k, the router should look up the entry (k, d) of the routing table, where d is the destination address, to determine the next node to route the packet to.

Default routing schemes built in the network simulator are as follows.

- Line: shortest-path routing that is deadlock free.
- Ring: shortest-path routing that may cause network deadlock. Deadlock-free non-minimal deterministic routing [15] is also supported.
- Mesh: XY-routing that is deadlock free.
- Torus: dimension-ordered routing. Along each dimension, there are two choices. The default mapping selects the shortest path. This scheme may cause deadlock. We also support deadlock-free non-minimal deterministic routing for torus [15].
- Hypercube: E-cube routing that is deadlock free.

Users can specify their own routing scheme by filling in the routing table with appropriate information. The routing information can be read from a file and stored in the routing table.

5.3 Network Simulation Algorithm

5.3.1 Overall Algorithm

System Quantum

Execution time of the parallel program is divided into equal slices; each slice is called a *system quantum.* A system quantum consists of a *computation quantum* running in parallel with a *communication round.* This results from the assumption that routers have dedicated processing units and run in parallel with processing elements.

The duration of a communication round is *flit latency* that a non-header flit takes to move from one node to an adjacent node. Therefore during a communication round, unblocked flits (if any) are moved forward by one link. Let the flit latency be $flit\_latency$, the duration of a system quantum be t_s, and the duration of a computation quantum be t_{cem}. We have $t_s = t_{cem} = flit\_latency$, since the computation quantum and the communication round are considered to run in parallel.

Let q_{cem} be

$$q_{cem} = t_{cem}/t_{clock} = flit\_latency/t_{clock}$$

where t_{clock} is the clock cycle of the processing elements. In every system quantum, the CEM runs for q_{cem} clock cycles (a computation quantum), and unblocked flits are advanced by one link (a communication round). For example, if $q_{cem} = 5$, the CEM executes for five clock cycles and the network manager advances unblocked flits by one link. In this case, the processing elements are considered very fast. If the network speed is high, we may have $q_{cem} = 0.25$. In this case, the CEM executes for one clock cycle every time the network manager runs four communication rounds.

The overall algorithm of the simulation is:

```
while (parallel program not finished)
{
  run one computation quantum; /*during this quantum, new messages to
                   be sent are inserted into the list of new messages*/
  run one communication round; /*considered to be running in parallel
                                          with the computation quantum*/
{
```

Computation Quantum

During a computation quantum, the CEM (Code Execution Module) takes control of the simulation. It executes parallel program code. If there are messages to be sent, the CEM passes message information to the network manager. The network manager then calculates the actual time to inject the message into the network. The injection time includes message and packet startup overheads. The corresponding *message* structure is then inserted into the list of new messages which is ordered by injection times of messages.

After the computation quantum expires, the control of the simulator is passed to the network manager that will run a communication quantum.

Communication Quantum

The core of a communication round is the communication step in which unblocked flits are advanced by one hop. Before the communication step is the message injection phase during which the network manager scans the list of new messages. If the injection time of a new message has come, the message is removed from the list of new messages and appended to the list of active messages. The network manager also cuts the message into packets and appends the packets to the list of active packets, preparing for routing.

To prepare for a communication step, the network manager first increments the network timestamp generator *clock_wh* that will be used for link scheduling. The network manager then runs one communication step to advance unblocked flits by one link. When all packets have been scheduled for moving, the network manager checks if routing deadlock has occurred. If so, the program execution is aborted. Otherwise, the network manager passes control to the CEM that will start the next computation quantum. The algorithm of a communication round is as follows.

```
communication_round
{
  inject timed-out new messages into network;
  increment timestamp generator clock_wh; /*used for link scheduling*/
  run a communication step; /*advance unblocked flits by one link*/
  check for deadlock;
}
```

A communication step for moving unblocked flits forward by one hop is described in the next subsection.

5.3.2 Communication Step

Hardware Implementation

The pipelining of successive flits in a packet can be done synchronously or asynchronously. Using *synchronous pipelining*, the network needs a network clock that is broadcast to all nodes. As the clock advances by one clock cycle, unblocked flits move forward by one link. Synchronous pipelining is simple to implement, but in general slower than asynchronous pipelining.

Asynchronous pipelining can be implemented using a handshaking protocol between adjacent routers [16]. This handshaking protocol requires a single-bit request/acknowledge (R/A) between every two adjacent routers in addition to the data channel. An example is given in Figure 46. The R/A line can be lowered only by the receiving router B to signal that B has available buffer for a new incoming flit. The R/A line can be raised only by the sending router A to indicate that A is transmitting a flit over the link. When B is ready to receive a flit, it lowers the R/A line. When A is ready to send, it raises the R/A line to high and transmits the flit over the link. While the flit is being received by B, the R/A line is kept high. After the flit is removed from B's buffer, the R/A line is lowered again and the cycle repeats. Asynchronous pipelining can be faster than synchronous pipelining, but it is more complex to implement.

Chittor's Implementation

In this simulator, messages are not packetized. The simulator does not support virtual channels. The flit train of a message is routed by maintaining the position of the header flit and the tail flit. Flits belonging to a message always occupy consecutive lanes.

Active messages are messages which are being advanced as opposed to blocked messages which are queued at routers waiting for the corresponding links to be released. A message whose header flit has reached the destination node is always active because the message does not need to request any more links and thus cannot be

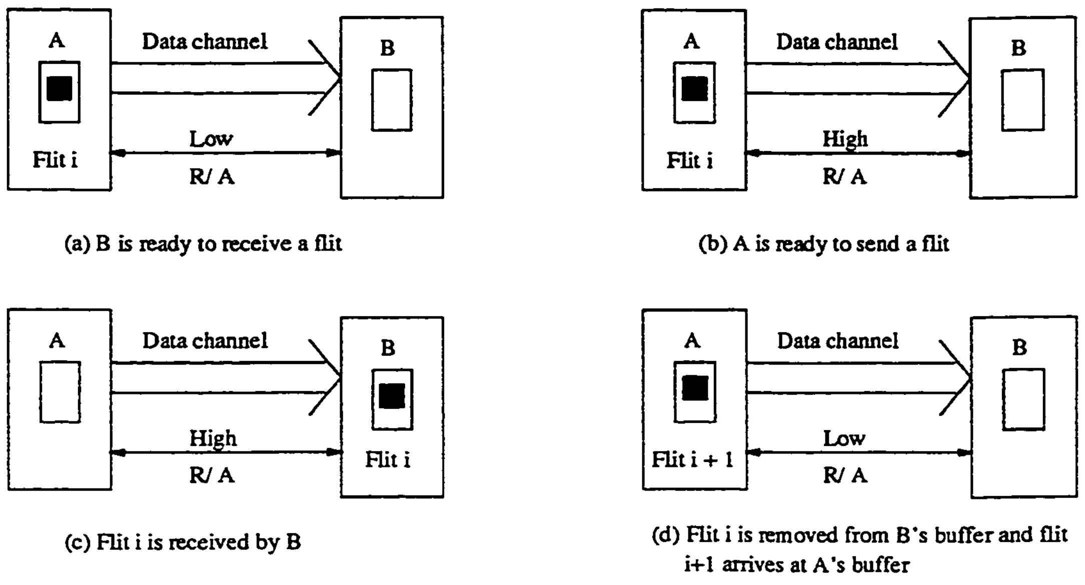

A
Data channel
B
Low
Flit i
R/ A
(a) B is ready to receive a flit
A
Data channel
B
High
Flit i
R/ A
(b) A is ready to send a flit
A
Data channel
B
High
R/ A
Flit i
(c) Flit i is received by B
A
Data channel
B
Low
Flit i + 1
R/ A
(d) Flit i is removed from B's buffer and flit i+1 arrives at A's buffer

Figure 46: Handshaking protocol between two adjacent routers

blocked.

A communication step which advances unblocked flits by one link consists of three phases. In the first phase, every active message whose header flit has not reached the destination computes the next node on the path. The message structure is then appended to the FIFO queue of the router at the node. This represents a request for the next link.

In the second phase, the above requests are processed to allocate links to requesting messages. For every message m in the list of active messages, if the requested link is currently busy or given to another message (i.e. the first message of the FIFO queue), m is blocked waiting in the FIFO queue for the link to be released. Thus m is removed from the list of active messages. However, if m is given the link (i.e. m is the first message of the FIFO queue), m is removed from the queue and the link status is set to *Busy* to indicate that the link has been allocated.

After the second phase, messages remaining in the list of active messages are those that have been granted a header link and those whose header flits have arrived at the destinations. Blocked messages were already removed from the list of active messages during phase 2 and are now waiting in router FIFO queues.

In the last phase, messages in the list of active messages are advanced by one link.

```
Phase 1: Messages request next links on their paths
  for every message m in the list of active messages
    if the head flit has not reached the destination
      { determine the next link;
        append message m to the FIFO queue at the next link;
      }
Phase 2: Requests are processed to allocate links to requesting messages
  for every message m in the list of active messages
    if the head flit has not reached the destination
      if the requested link L has been busy or given to another message
        remove message m from the list of active messages;
      else { remove message m from the FIFO queue at the next link;
             set status of link L to Busy;    /*link is given to this message*/
           }
  /* After phase 2, messages remaining in the list of active messages will be
   * advanced by one link. Blocked messages were already removed from the
   * list of active messages and are staying at respective FIFO queues */
Phase 3: Unblocked messages are advanced by one link
  for every message m in the list of active messages
    { advance message m by one link;
      if the tail flit of m releases a link and the link FIFO queue is not empty
        append the first waiting message to the list of active messages;
                  /*the link is freed, so unblock a waiting message*/
      if the whole message m has arrived at the destination
        remove message m from the list of active messages;
    }
```

Figure 47: Chittor's algorithm for one communication step

The positions of the header flits and tail flits are recorded. When a tail flit leaves a link, the link is freed and the FIFO queue at the corresponding router is examined. If there are waiting messages, the first message in the queue is given the link, removed from the queue and appended to the list of active messages. If the tail flit arrives at the destination (i.e. the whole message has been received), the message is removed from the list of active messages.

The algorithm of a communication step is summarized in Figure 47.

Dally's Implementation

This simulator implements virtual channels in details. Messages are not packetized. Newly sent messages are appended to the list of messages (*message_list*). There is a temporary list of links (*temp_list*). When a message makes a request at a link (either for a free lane or for link bandwidth allocation), if the link is currently not on this list, the link is added to the list. Only requested links need allocation/arbitration: the links in the *temp_list* are examined to process the requests.

A communication step to advance unblocked flits by one link consists of four phases. In phase 1, every message whose header flit has not reached the destination requests a free lane on the next link. The message first determines the ID of the next link on its path. It then appends a request for a free lane to the *lane allocation queue* at the next link. If this link is currently not in the *temp_list*, it is added to the list. At the end of phase 1, the *temp_list* contains the links requested by messages for free lanes.

In phase 2, the *temp_list* is traversed to process the requests for lanes. Every link in the *temp_list* grants its free lanes to requesting messages using a pre-determined lane allocation scheme. If there are more requests than free lanes, outstanding requests are discarded. The owner messages of these requests will repeat phase 1 in the next communication step, attempting to get free lanes for their header flits. At the end of phase 2, the *temp_list* is emptied for use in phase 3.

In phase 3, every message requests link bandwidth from the links on its path to advance the flits. For each link *occupied_link* on the current path of the message, a request for link bandwidth is appended to the *link bandwidth allocation queue* at *occupied_link*. The link is then added to the *temp_list* if it is currently not in the list. At the end of phase 3, the *temp_list* contains the links whose bandwidth is demanded for advancing flits.

The *temp_list* is scanned in phase 4 to allocate link bandwidth to requesting messages. Every link in the *temp_list* is scheduled using a pre-determined link bandwidth allocation scheme to decide which lane will use the link. Only lanes that have a non-empty sending buffer and a non-full receiving buffer can take part in the scheduling. Assuming that lane *sched_lane* is granted the link bandwidth in this step, the flit in the previous lane is then moved into lane *sched_lane*. At the end of phase 4, the *temp_list* is freed.

The algorithm of the communication step is shown in Figure 48.

CPSS Network Implementation

We wanted to eliminate the implementation with so many phases. The simulation time would be intolerable when the network is combined with the code execution module.

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

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

```
communication_step
{
  for each active packet     /* phase 1 */
     if header flit has not reached destination yet
         request the next head lane;

  for each active packet p    /* phase 2 */
    for each lane L on the current path of  p
      /*starting from  header lane going backward to tail lane*/
    { if the corresponding link has not been scheduled in this
                 communication step /* check field schedTime */
         schedule the link;
```

```
Phase 1: Every message requests a free lanes on the next link
  /*Pre-condition: the temp_list is empty*/
  for each message in the message_list
    if the header flit has not reached the destination
      { determine the next link (next_link) on the path;
        append a request for a free lane to the queue at next_link;
        if next_link is not in the temp_list
          add next_link to the temp_list;
      }
Phase 2: Requested links allocate free lanes to requesting messages
  for each link in the temp_list
     allocate free lanes to requesting messages queued at the link;
          /*If some message is not given a free lane, its request is discarded*/
          /*The message will queue a new request in the next communication step*/
  free the temp_list;
  /*Post-condition: the temp_list is empty*/
Phase 3: Flits of messages request link bandwidth to move forward
  /*Pre-condition: the temp_list is empty*/
  for each message in the message_list
    for each link occ_link occupied by the message
      { append a request for link bandwidth to the queue at occ_link;
        if occ_link is not in the temp_list
          add occ_link to the temp_list;
      }
Phase 4: Links are scheduled to advance flits
  for each link on the temp_list
    { schedule the link; the bandwidth is then given to lane sched_lane;
      move the flit in the previous lane into lane sched_lane;
    }
  free the temp_list;
  /*Post-condition: the temp_list is empty*/
```

Figure 48: Dally's algorithm for one communication step

```
        if the lane scheduled to use the link is lane L
                                        /* check field scheduledLane */
            advance the flit in lane L;
    }
}
```

Using this algorithm, not all links need to be scheduled during a communication step. In fact, only occupied links are required to be scheduled.

The issue now is how a packet can know whether a link has already been scheduled by another packet in the same communication step to avoid a second scheduling on that link. A link is said to be successfully scheduled in a communication step if there exists at least one lane which is able to use the link in this step (i.e. the lane has a non-empty buffer in the source node and a non-full buffer in the receiver node).

When a link is successfully scheduled, its *schedTime* field is updated with the current value of the timestamp generator to indicate that the link has been scheduled in this communication step. At the same time, field *scheduledLane* is recorded with the ID of the lane that is allocated the link bandwidth in this communication step. When the network manager tries to advance another packet q using the same link, it examines field *schedTime* for the scheduling timestamp and sees that it does not have to schedule the link a second time in this communication step. The timestamp generator is incremented after every communication step for this purpose. Packet q will look up field *scheduledLane* to decide whether its flit is allowed to use the link in this step.

If the scheduling fails (i.e. no lane can be found at this moment for using the link bandwidth), field *schedTime* is not updated. Some lane which cannot be scheduled now may be able to use the link later (within the same communication step) after the flits in front of this lane have moved forward.

5.3.3 Link Bandwidth Allocation

A physical link is time-shared by several lanes (virtual channels) to maximize link utilization. The scheduling is done in a round-robin fashion.

Definitions

A lane is *not full* if the number of flits currently stored in this lane is less than the buffer size.

A lane is *not empty* if there is at least one flit buffered in this lane.

The *first lane* of a packet is the lane on the path of the packet connected to the source node.

The *destination lane* of a packet is the lane on the path of the packet connected to the destination node.

A flit is considered to arrive at the destination node if it has arrived at the destination lane.

To measure whether a lane is not full or not empty, a counter is used, which is field *num_flits* of the *lane* structure. Field *num_flits* associated with every lane records the number of flits currently stored in the lane. Let *buffer_size* be the size of the flit buffer at each lane. In general, it should be that $0 \leq num\_flits \leq buffer\_size$.

To save space, field *num_flits* of a destination lane is also used to count the number of flits of a packet which have arrived at a destination lane (and thus at the destination node). Consequently, for destination lanes, the value of *num_flits* can reach the packet size which is usually larger than the buffer size. For this reason, we do not perform the checking *lane not full* on destination lanes.

Let *packet_size* be the packet size. A packet is considered to be received completely by the destination node if its destination lane has $num\_flits \geq packet\_size$. In summary,

- $0 \leq num\_flits \leq buffer\_size$ for non-destination lanes
- $0 \leq num\_flits \leq packet\_size$ for destination lanes

Schedulability

A lane is *schedulable* if the flit (or one of the flits) in the previous lane can be advanced to this lane. So the conditions should be: the current lane is not full and the previous lane is not empty.

The above conditions cannot always be tested since there are special cases:

- The *lane not full* checking cannot be done on destination lanes due to the dual purposes of field *num_flits* as just mentioned above.

- First lanes do not have a previous lane because unsent flits are buffered at the source node.

If a lane satisfies one of the following cases, it is schedulable.

1. - The current lane is not a destination lane and not full.
   - The previous lane is not nil and not empty.

   This is the general case described above in the definition of a schedulable lane.

2. - The current lane is not a destination lane, not full and is a first lane.
   - The previous lane is nil.

   An example of this case is when the current lane is adjacent to the source node. The lane is scheduled to transfer the flits buffered at the source to this lane.

3. - The current lane is a destination lane.
   - The previous lane is not nil and not empty.

   Since the current lane is a destination lane, we do not check for *lane not full.*

4. - The lane is a destination lane.
   - The previous lane is nil.

   An example of this case is when the source and the destination are adjacent nodes. The lane is scheduled to transfer a flit buffered at the source to the destination node.

A free lane is of course not schedulable. Only schedulable lanes of a link may participate in the scheduling in order to obtain the link bandwidth to advance one flit. The purpose of this implementation is to maximize link utilization.

5.3.4 Approximate Round-Robin Scheduling for Link Bandwidth Allocation

The CPSS employs round-robin scheduling with some *approximation* for link bandwidth allocation. In this sub-section, we explain why such approximation is needed in our network simulation.

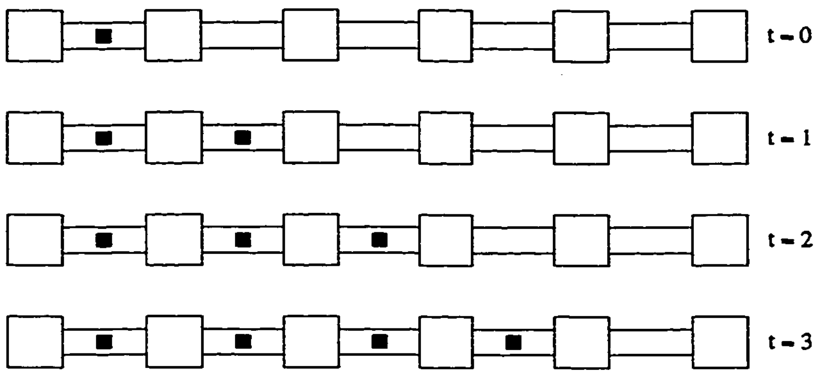
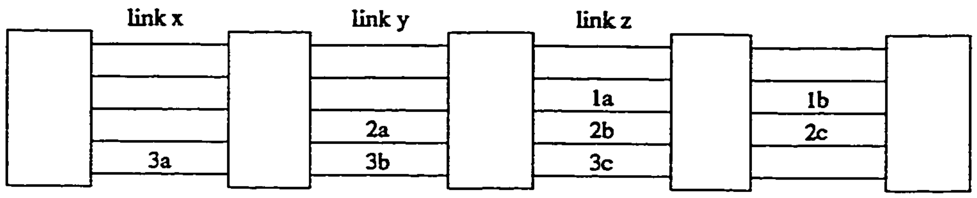


Figure 49: Chittor's implementation of flit movement

Motivation: To Speed Up Simulation Time

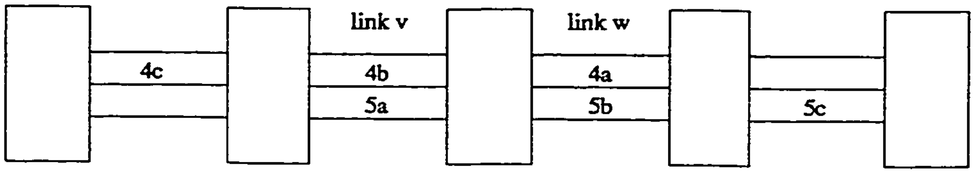
Chittor's simulator and Dally's simulator both support round-robin scheduling for link bandwidth allocation. Chittor's simulator does not implement virtual channels. Message routing simulation is thus straightforward: only the header flit and the tail flit need to be kept track of. When the header flit moves forward by one link, the following flits advance accordingly. As a flit moves forward, it leaves behind an empty buffer so that the following flit can move into this buffer. Flits of a packet are always buffered in consecutive lanes; there are no empty holes between flits of a packet (Figure 49). In this simulator, a source node injects the flits of a packet at the rate of one flit per time unit (flit latency unit). Similarly, a destination node accepts flits at the rate of one flit per time unit as well.

When virtual channels are implemented, link scheduling is required to allow virtual channels to time-share physical links. In this case, scheduling one link may depend on the scheduling result of another link. An example is shown in Figure 50a. We assume that each physical link is shared by four lanes, and the lane buffer size is one flit. In the figure, three packets 1, 2 and 3 are being routed on three paths whose header lanes are $1a$, $2a$ and $3a$ respectively. We want to schedule link z that is shared by lanes $1a$, $2b$ and $3c$. To maximize link utilization, only schedulable lanes are allowed to participate in the scheduling. However, we do not know whether lane $2b$ is schedulable or not until lane $2a$ is scheduled, at which time we know if the flit in lane $2a$ can move forward in this communication step. Thus we need to schedule link y before link z. Similarly, to know if lane $3b$ of link y is schedulable or not, we



1a, 2a, 3a: header lanes of packets 1, 2, and 3, respectively

a) Noncyclic link scheduling dependency



4a, 5a: header lanes of packets 4 and 5, respectively

b) Cyclic link scheduling dependency

Figure 50: Link scheduling dependency

have to examine lane $3a$ first. This requires to schedule link x before link y. The dependency chain stops here because lane $3a$ exclusively occupies link x. Therefore the link scheduling order is x, y, z.

Establishing link scheduling order would require

- searching for packets with lanes which are schedulable and exclusively occupy corresponding links, and
- ordering the packets to form a dependency chain.

This implementation would slow down the simulation seriously.

Dally's simulator overcomes this problem by allowing holes in between the flit train of a packet even if there is only one packet to be routed in the network. That is, the flits of a packet are not occupying consecutive lanes on the path even if there is no link bandwidth contention. Figure 51 illustrates the scenario. In this implementation, the flits of a packet are injected by the source node at the rate of one flit every two time units. A destination node also receives flits at the rate of one flit every two time units. In order for a packet to reach the destination, Dally's simulator must run approximately twice as many communication steps as Chittor's simulator. This would

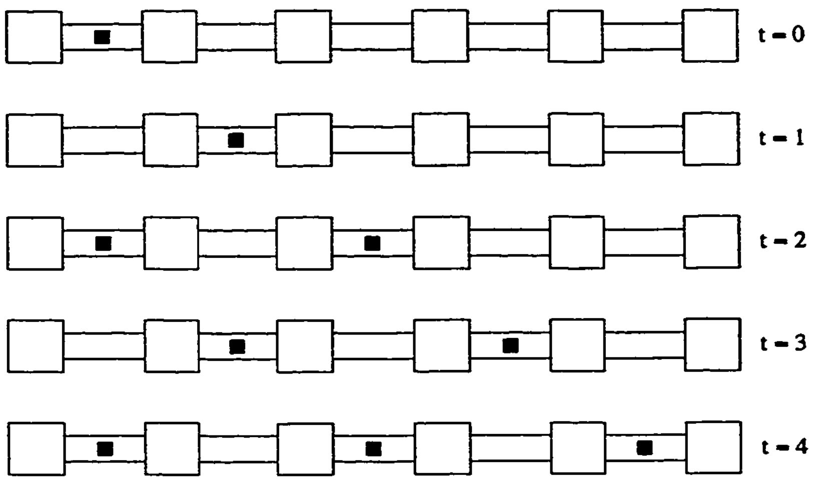


Figure 51: Dally's implementation of flit movement

increase simulation time considerably. Therefore we adopt Chittor's implementation of flit movement and add support for virtual channels. Note that holes may exist in between the flit train of a packet p if some links on p's path are time-shared by virtual channels. This scenario is inevitable in the presence of virtual channels. In Chittor's simulator, the flits of a packet are always consecutive because this simulator does not support virtual channels.

To solve the issue of link scheduling dependency and to minimize simulation time, we employ round-robin scheduling with some approximation.

Link Scheduling Dependency

Scheduling one link may depend on the scheduling result of another link. One of the following two cases may happen:

1. The dependency chain ends when there is a packet with one lane which is schedulable and exclusively occupies the corresponding link. The flit in this lane should be advanced first to allow other links to be scheduled. Scheduling of other links should then be done following the dependency chain (in the reverse order).

   This case is illustrated in Figure 50a, and has been discussed above. Establishing link scheduling order would slow down the simulation time significantly.

2. We may not be able to find a packet with a lane which is schedulable and exclusively occupies the corresponding link. In this case we encounter a cyclic dependency. In order to break the cycle, we should give any schedulable lane its corresponding link bandwidth to advance the flit stored in this lane, hoping to break the cycle. In this case, the chosen lane may not have been selected by the strict round-robin rule.

   An example of this case is shown in Figure 50b. In this figure, two packets 4 and 5 are being routed. Their header lanes are $4a$ and $5a$ respectively. In order to schedule link w shared by lanes $4a$ and $5b$, we must determine the schedulability of lane $5b$, which depends on whether lane $5a$ will be allocated link v. Thus we must schedule link v before link w. To schedule link v, we must know if lane $4b$ is schedulable. This knowledge then depends on the scheduling of link w to determine if lane $4a$ will be allowed to use link w in this communication step. Thus link w must be scheduled before link v. This is a cyclic dependency.

In both cases, round-robin scheduling with some approximation is necessary to either minimize the simulation time (case 1) or to break the dependency cycle (case 2).

Definition of Approximate Round-Robin Scheduling

If

- a lane a is selected by the strict round-robin rule and is not schedulable at this point in time, and
- there exists a lane b down the round-robin circle which shares the same link and is schedulable at this point in time

then the link is given to b in this communication step (although a was selected by the strict round-robin rule).

If we used the strict round-robin rule, it may happen that lane a is eventually not schedulable when the packet occupying a is moving. In this case, the link would waste one network clock cycle. We should give the link to a schedulable lane (e.g. lane b) to maximize link utilization. Approximate round-robin scheduling thus helps to improve link utilization as well.

In summary, the advantages of approximate round-robin scheduling for link bandwidth allocation are: to speed up the simulation time, to break the scheduling dependency cycle (if any), and to enhance link utilization.

5.3.5 Header Flit Overheads

We have so far considered only the movement of non-header flits. For header flits, we must take into account the *header overhead* that is the sum of *router decision time* and *lane allocation time* (section 5.1.9). In the network simulator, the header overhead is a definable parameter (*headerOverhead*) and a multiple of flit latency unit. Thus the header overhead takes integer values greater than 0.

Field *headerOverheadAcc* of the *packet* structure is used to simulate the overheads incurred by header flits. When a header flit is allocated the next free lane, field *headerOverheadAcc* of the corresponding packet is initialized to 0. In each communication step, the network manager schedules links for moving unblocked flits by one hop as mentioned earlier. If the network manager sees an empty header lane (field $nbrFlits = 0$), this means that the lane has been reserved for a packet but its header flit has not been moved into the lane yet. In this case, the network manager compares field *headerOverheadAcc* of the corresponding packet against the value of parameter *headerOverhead*: if $headerOverheadAcc < headerOverhead$, then this header lane is not allowed to participate in the scheduling (i.e. the lane is not schedulable). At the same time, field *headerOverheadAcc* of the packet is incremented to count one more flit latency unit. After a number of increments, field *headerOverheadAcc* equals the value of parameter *headerOverhead*. Only then does this header lane become schedulable. The header flit is then treated as other non-header flits for advancing to the reserved lane.

5.4 Performance Comparison with Dally's Simulator

This section shows the performance comparison between our network simulator and Dally's simulator. We did not consider Chittor's simulator because it does not support virtual channels. Also, when the number of virtual channels per link is one, our simulator behaves as Chittor's.

Our goal is to compare the performance of the network portion of the CPSS with Dally's simulator. We carried out the experiments using the stand-alone wormhole-routed network simulator since otherwise run-time overheads of the code execution module and of vCode interpretation could affect the pure simulation time of the network.

The comparison results are shown in Table 1. In the table, the first, second and third columns indicate the topology, network size (number of nodes) and number of messages involved in each experiment respectively. The fourth and fifth columns show the simulation times (in seconds) obtained from our network simulator and Dally's respectively.

In the experiments, Dally's simulator uses strict round-robin scheduling while ours employs approximate round-robin scheduling. In each experiment, I/O time is excluded to obtain accurate simulation time. Messages are randomly generated. Each experiment (i.e. each line in Table 1) was run ten times and the resulting simulation time is the average of the ten runs. The results show that our simulator outperforms Dally's in all the experiments. This is due to the use of approximate round-robin scheduling for virtual channels (section 5.3.3), and the reduction in the number of phases involved in a communication step (section 5.3.2).

To obtain the results shown in Table 1, the parameters of the simulated wormhole-routed network were given the following typical values:

- network topology and size: as shown in Table 1
- number of lanes per link = 4
- bidirectional links and bidirectional lanes
- buffer size = 1 flit for networks of 256 nodes or less, and 2 flits for networks of more than 256 nodes
- packet size = 8 bytes
- flit size = 8 bits
- flit latency = 1 time unit
- message startup overhead = 10 time units

- packet startup overhead = 10 time units
- header overhead = 5 time units

We repeated the experiments using different sets of parameters. All the outcomes are consistent with the results given in Table 1.

Topology	Network Size	Number of Messages	Running Time (in seconds)	
			CPSS Network	Dally's Simulator
Line	50	100	1.20	1.75
		300	4.12	5.95
	100	200	5.30	6.67
		500	14.48	20.68
	300	500	104.60	135.15
		800	262.06	369.93
2D Mesh	64 (8x8)	100	0.22	0.45
		300	0.70	1.32
	100 (10x10)	200	0.60	1.23
		500	1.46	3.08
	225 (15x15)	300	1.35	3.50
		600	2.40	6.82
	400 (20x20)	500	5.85	14.03
		1,000	11.93	27.90
3D Mesh	64 (4x4x4)	100	0.18	0.53
		300	0.57	1.56
	125 (5x5x5)	300	0.65	2.51
		600	1.38	4.86
	343 (7x7x7)	500	3.37	13.08
		1,000	7.47	27.95
	512 (8x8x8)	600	4.22	18.58
		1,000	8.01	34.25
Hypercube	32	100	0.17	0.50
		300	0.55	1.43
	128	200	0.62	2.15
		500	1.45	5.05
	512	500	4.86	17.28
		1,000	8.75	36.49

Table 1: Performance comparison with Dally's simulator

Chapter 6

Optimal Program Mappings for Wormhole-Routed Networks

In this chapter, we present the one-to-one mappings among the most important topologies: lines, rings, hypercubes, square meshes and square tori. These topologies represent the communication structures of many applications in scientific computations as well as the topologies of many large-scale wormhole-routed networks [5].

Our mapping objective is to minimize the maximum *path contention level* (PCL) of a task graph. Path contention level of a path p connecting two communicating processes can be roughly defined as the number of other paths which share at least one link with path p. Path contention level represents the worst-case contention of a path. That is, it assumes that all competing paths are working at the same time. In reality, some of the competing paths may not route any messages at some time, depending on the application communication pattern. In any case, if we can minimize the path contention level of a path, the path will collide with other paths as less as possible. This effectively reduces blocking time of messages.

We first define the terminology and notations used in this chapter. The mapping functions are then presented and their optimality is proved. Finally, we provide experimental results which show that our mapping functions significantly outperform random mappings in terms of communication performance, especially on large networks.

6.1 Definitions and Notations

A task graph (G_t) is a directed graph that consists of a set of processes (V_t) and a set of communication channels (E_t). A communication channel represents message sends/receives between two communicating processes.

A system graph (G_s) is an undirected graph that consists of a set of processors (V_s) and a set of physical links (E_s).

A mapping function f maps the processes of G_t onto the processors of G_s, i.e., $f : V_t \rightarrow V_s$. A mapping is defined by a mapping function f and a deterministic routing algorithm R. Under a mapping, every communication channel $e \in G_t$ is mapped onto a unique path of the system graph.

The dilation cost of a mapping is the length of the longest path connecting any two communicating processes.

The path contention level (PCL) of a path p corresponding to a communication channel $e \in G_t$ is the number of paths that share at least one link with p. The PCL of the path p or the communication channel e is denoted by $PCL(p)$ or $PCL(e)$ respectively.

The contention c of a mapping is defined to be the maximum PCL of that mapping.

$$C = \max_{e \in E_t}\{PCL(e)\}$$

The link load of a physical link $i \in E_s$ is the number of paths which traverse link i and denoted by $LL(i)$.

The congestion l of a mapping is defined to be the maximum link load of that mapping.

$$l = \max_{e \in E_s}\{LL(e)\}$$

In the following sections, all variables assume non-negative integers unless otherwise stated. Given an integer $k \geq 1$, we use $[k]$ to denote the set $\{0, 1, ..., k-1\}$. Given two integers a and b, the notation $a \div b$ is equivalent to $\lfloor a/b \rfloor$. For both the task graph and the system graph, we use

- n to denote the total number of nodes the graph has;
- m to denote the number of nodes along each dimension of a square mesh or torus; so $n = m^2$ for a 2-D mesh or torus and $n = m^3$ for a 3-D mesh or torus;

- d to denote the dimension of a hypercube. So $n = 2^d$.

In a line or a ring, the nodes are numbered from 0 to $n-1$. In a 2-D mesh (or torus), each node is identified by a tuple (x, y) where x and y indicate the row and the column of the node in the mesh (or torus) respectively. Similarly, each node of a 3-D mesh (or torus) is denoted by the tuple (x, y, z) where x, y and z are the row number, the column number and the plane number of the node in the mesh (or torus) respectively. Note that x, y and z belong to $[m]$.

The mappings are one-to-one and every process communicates with all of its neighboring processes in the task graph. We also assume that all physical links of the system graph are bidirectional.

6.2 Mapping Functions

6.2.1 Review of Existing Mappings

Proposition 1 *Any one-to-one mapping function with unit dilation cost has contention 0.*

A mapping function with unit dilation cost implies that any pair of neighboring processes is mapped onto a distinct pair of adjacent processors. So there will be no physical link conflicts in the wormhole-routed network. [6] and [7] presented several mappings with unit dilation cost. By proposition 1, these mappings have contention 0.

In the following sub-sections, we propose new mapping functions which minimize the contentions of task graphs.

6.2.2 Mapping of a Ring onto a Line

Proposition 2 *The contention C of a mapping with dilation cost d and congestion l is bound by $l - 1 \leq C \leq d(l-1)$.*

Proof:

Lower Bound: Since the congestion is l, the contention cannot be less than $l-1$.

Upper Bound: The longest path p (corresponding to a communication channel of the task graph) traverses d links, each having load l in the worst case. On each link, the given path may compete with $l-1$ other paths. Thus, $PCL(p) \leq d(l-1)$.

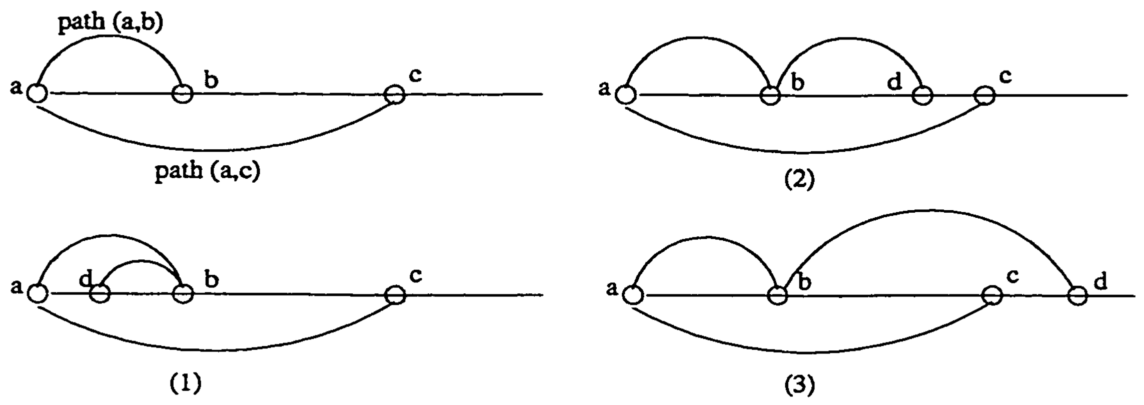


Figure 52: Illustration for the proof of Proposition 3

Proposition 3 *The contention of any mapping of a ring onto a line is at least 2.*

Proof: Given any mapping π from the ring to the line, let a be the process of the ring mapped onto the first processor of the line, i.e., $\pi(a) = 0$. Let the two neighbors of a in the ring be b and c respectively. Without loss of generality, assume that $\pi(b) < \pi(c)$. Let d be the another neighbor of b. $\pi(d)$ can be either between $\pi(a)$ and $\pi(b)$, or between $\pi(b)$ and $\pi(c)$, or $\pi(c) < \pi(d)$. In either case the three paths corresponding to the communication channels (a, b), (a, c), and (b, d) overlap on at least one link, as illustrated in Figure 52. Therefore the proposition is true.

The following function $x : [n] \rightarrow [n]$ maps a ring onto a line:

$$x(u) = \begin{cases} 2u & \text{if } u < (n+1) \div 2 \\ 2(n-u)-1 & \text{otherwise} \end{cases}$$

It is easy to verify that this mapping has dilation cost 2 and congestion 2. By Proposition 2, the contention C of the mapping is bound by $1 \leq C \leq 2$. By Proposition 3, we have $C = 2$, which is also the least contention we can obtain when mapping a ring onto a line. This mapping is thus optimal in terms of contention.

6.2.3 Mapping of a Ring onto a 2-D Mesh

Given a process $u \in [n]$, we define the tuple $(x(u), y(u))$ denoting the processor to which process u will be assigned, where $x(u) \in [m]$ and $y(u) \in [m]$. When m is even, the mapping is defined in [6] and has unit dilation cost. Its contention is thus 0 by Proposition 1. When m is odd and $m > 3$, the mapping is defined as follows.

Let $\alpha = (m-4)(m-1) + 2m$

Case 1: $0 \leq u < m : x(u) = u,\ y(u) = 0$

Case 2: $m \leq u < 2m-1 : x(u) = m-1,\ y(u) = u-(m-1)$

Case 3: $2m-1 \leq u < \alpha$: Define $v = u - (2m-1)$

$$y(u) = (m-1) - (v \div (m-1))$$

$$x(u) = \begin{cases} v \bmod (m-1) & \text{if } y \text{ odd} \\ (m-2) - (v \bmod (m-1)) & \text{if } y \text{ even} \end{cases}$$

Case 4: $x(\alpha) = 0,\ y(\alpha) = 2$

Case 5: $x(\alpha+1) = 1,\ y(\alpha+1) = 2$

Case 6: $\alpha + 2 \leq u < \alpha + m$: $x(u) = u - (\alpha+1),\ y = 3$

Case 7: $\alpha + m \leq u \leq m^2 - 3$: Define $k = u - (\alpha + m)$

$$x(u) = (m-2) - (k \div 2)$$

$$y(u) = \begin{cases} 2 & \text{if } k \bmod 4 = 0 \text{ or } (k+1) \bmod 4 = 0 \\ 1 & \text{otherwise} \end{cases}$$

Case 8: $x(m^2-2) = 1,\ y(m^2-2) = 1$

Case 9: $x(m^2-1) = 0,\ y(m^2-1) = 1$

The contention of this mapping is 0. It is easy to verify from the mapping function that each pair of neighboring processes, except the pair (m^2-3, m^2-2), is mapped onto a distinct pair of adjacent processors (Figure 53). On the other hand, since the physical links are bidirectional, communications between processes m^2-3 and m^2-2 take the path ((2,2)-(1,2)-(1,1)) which does not compete with any of the other paths. The contention of the mapping is thus 0.

6.2.4 Mapping of a Ring onto a 3-D Mesh

A process $u \in [n]$ is mapped to processor $(x(u), y(u), z(u))$, where $x(u)$,$y(u)$ and $z(u)$ are in $[m]$. When m is even, the mapping is defined in [6] and has unit dilation cost. Its contention is thus 0 by Proposition 1. When m is odd, the mapping is defined as follows.

Case 1: $0 \leq u < m(m^2-1)$:

$$z = u \div (m^2-1)$$

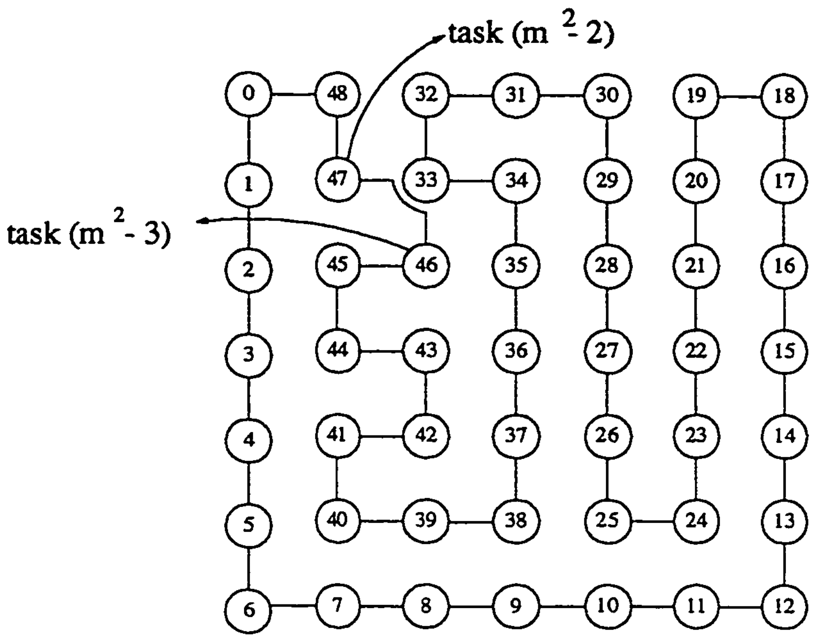


Figure 53: Mapping of a ring onto a 2-D mesh (m=7)

Let

$$t = \begin{cases} u - z(m^2-1) & \text{if } z \text{ even} \\ (m^2-2) - (u - z(m^2-1)) & \text{if } z \text{ odd} \end{cases}$$

Case 1a: $0 \le t < m$: $x(u) = t$, $y(u) = 0$

Case 1b: $m \le t < (m-1)^2$: Define $g = (t-1) \div (m-1)$

$$x(u) = m - g$$

$$y(u) = \begin{cases} t - g(m-1) & \text{if } g \text{ odd} \\ (g+1)(m-1) - (t-1) & \text{if } g \text{ even} \end{cases}$$

Case 1c: $(m-1)^2 \le t < m^2 - 1$: Define $k = t - (m-1)^2 - 1$

$$x(u) = \begin{cases} 1 & \text{if } k \bmod 4 = 0 \text{ or } (k+1) \bmod 4 = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$y(u) = (m-1) - (k \div 2)$$

Case 2: $m(m^2-1) \le u < m^3$: $x(u) = 1, y(u) = 1, z(u) = (m^3-1) - u$

The contention of this mapping is 0. It is easy to verify from the mapping function that each pair of neighboring processes, except the pair $(m^3-1, 0)$, is mapped onto a

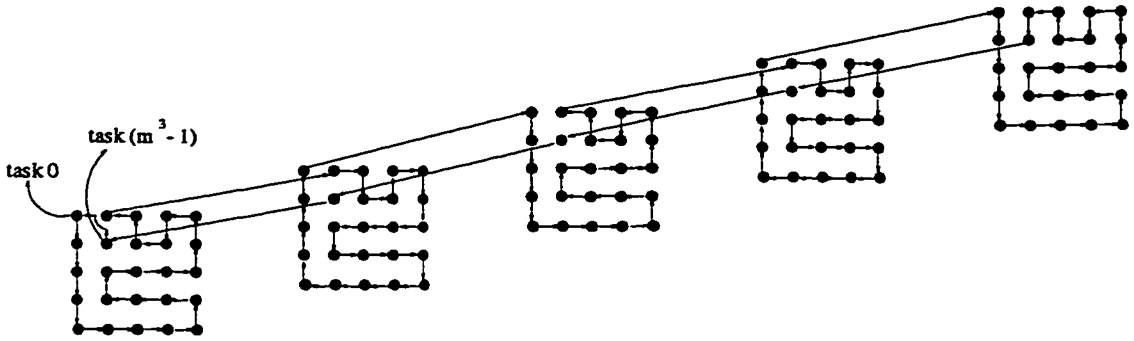


Figure 54: Mapping of a ring onto a 3-D mesh (m=5)

distinct pair of adjacent processors (Figure 54). On the other hand, since the physical links are bidirectional, communications between processes $m^3 - 1$ and 0 take the path ((1,1,0)-(0,1,0)-(0,0,0)) which does not compete with any of the other paths. The contention of the mapping is thus 0.

6.2.5 Mapping of a 2-D Torus onto a 2-D Mesh

Proposition 4 *The contention of any mapping of a 2-D torus onto a 2-D mesh is at least 2, assuming that XY routing is used.*

Proof: Let X and Y be the first and second dimensions, and $(X(e), Y(e))$ be the processor onto which a process e is mapped. Assume that every message goes first along X and then along Y. Let a be the process assigned to the upper left processor, i.e., $X(a) = 0$ and $Y(a) = 0$. Since every process communicates with alll of its neighbors in the task graph, assume that a broadcasts some messages to its four neighbors. The four neighbors of a can be mapped as one of the following cases.

- More than two neighbors of a are on column 0 (Figure 55(a) and (b)). In this case, the link load of link $l2$ is $LL(l_2) \geq 3$. So contention $C \geq 2$ by Proposition 2.
- More than two neighbors of a are not on column 0 (Figure 55(c) and (d)). Similarly, $LL(l_1) \geq 3$. So $C \geq 2$.
- a has exactly two neighbors on column 0. Let them be b and e (Figure 55(e)).

Without loss of generality, assume that $Y(a) < Y(b) < Y(e)$, and that b broadcasts some messages to its neighbors. b has three other neighbors besides a, and they are mapped as one of the following two cases.

- These three neighbors of b are not on column 0. In this case, $LL(l_3) \geq 3$ (Figure 55(f)). So $C \geq 2$ by Proposition 2.
- At least one of these three neighbors of b is on column 0. Let d be one such neighbor. $Y(d)$ can be either between $Y(a)$ and $Y(b)$, or between $Y(b)$ and $Y(e)$, or $Y(e) < Y(d)$. In either case, the three paths corresponding to the communication channels (a, b), (a, e), and (b, d) overlap on at least one link, as illustrated in Figure 55(f)(see also the proof of Proposition 3). So $PCL(a, e) \geq 2$.

Therefore in all cases, $C \geq 2$. We would obtain the same result by a similar proof if messages go first along Y and then along X.

Given a process (x, y), the processor $(f_x(x, y), f_y(x, y))$ on which process (x, y) is mapped is defined by

$$f_x(x,y) = \begin{cases} 2x & \text{if } x < (m+1) \div 2 \\ 2(m-x)-1 & \text{otherwise} \end{cases}$$

$$f_y(x,y) = \begin{cases} 2y & \text{if } y < (m+1) \div 2 \\ 2(m-y)-1 & \text{otherwise} \end{cases}$$

where $x, y, f_x(x, y)$ and $f_y(x, y)$ are in $[m]$.

It is easy to verify that this mapping has dilation cost 2 and congestion 2. By Proposition 2, the contention C of the mapping is bound by $1 \leq C \leq 2$. By Proposition 4, C is 2, which is also the least contention we can obtain when mapping a ring onto a line. This mapping is thus optimal in terms of contention.

6.2.6 Mapping of a 3-D Torus onto a 3-D Mesh

Proposition 5 *The contention of any mapping of a 3-D torus onto a 3-D mesh is at least 2, assuming that XYZ routing is used.*

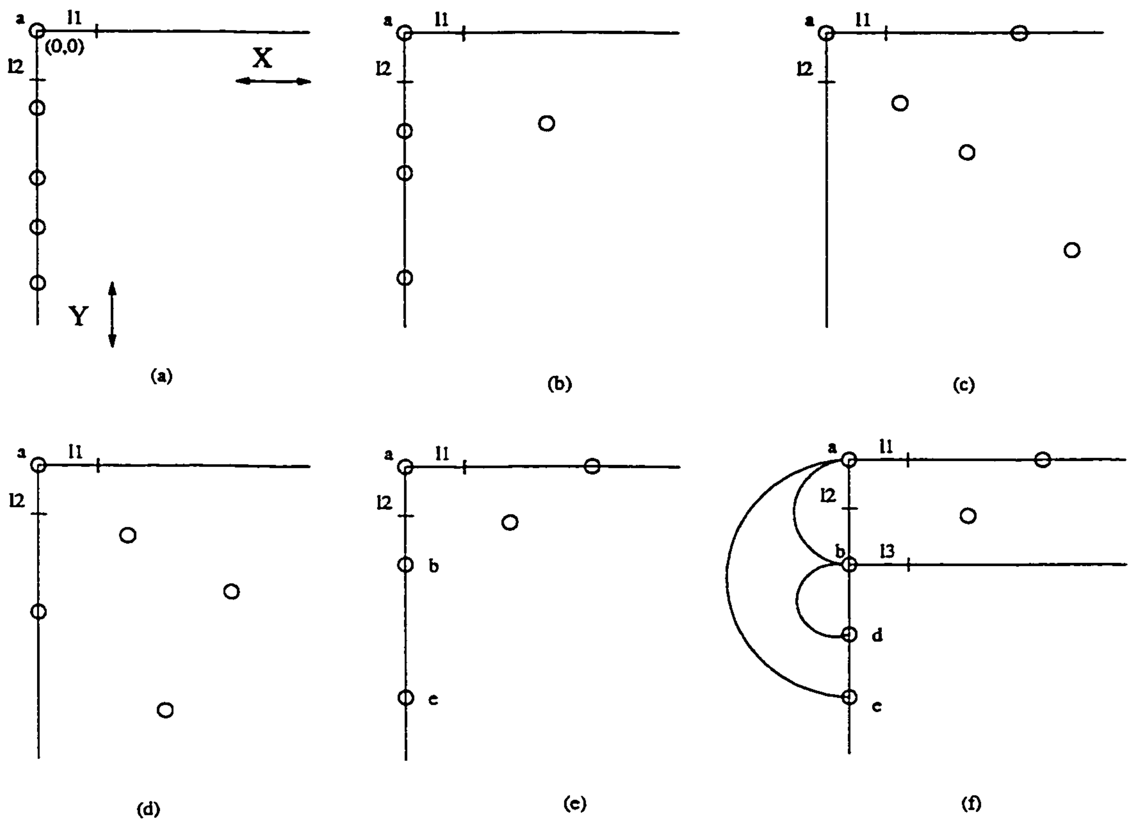


Figure 55: Illustration for the proof of Proposition 4

The proof of this proposition is similar to that of Proposition 4

A task graph represented by tuple (x, y, z) is assigned to processor $(f_x(x, y, z), f_y(x, y, z), f_z(x, y, z))$ by the following mapping:

$$f_x(x,y,z) = \begin{cases} 2x & \text{if } x < (m+1) \div 2 \\ 2(m-x)-1 & \text{otherwise} \end{cases}$$

$$f_y(x,y,z) = \begin{cases} 2y & \text{if } y < (m+1) \div 2 \\ 2(m-y)-1 & \text{otherwise} \end{cases}$$

$$f_z(x,y,z) = \begin{cases} 2z & \text{if } z < (m+1) \div 2 \\ 2(m-z)-1 & \text{otherwise} \end{cases}$$

where $x, y, z, f_x(x, y, z), f_y(x, y, z)$ and $f_z(x, y, z)$ are in $[m]$.

It is easy to verify that this mapping has dilation cost 2 and congestion 2. By Proposition 2, the contention C of the mapping is bound by $1 \leq C \leq 2$. By Proposition 5, C is 2, which is also the least contention we can obtain when mapping a ring onto a line. This mapping is thus optimal in terms of contention.

6.3 Experimental Results

We selected three parallel programs: Exchange Sort, N-Body and Matrix Multiplication [25]. Their task graph topologies are line, ring and 2-D torus respectively. Each program is mapped onto different topologies of various sizes under corresponding optimal mappings and random mappings, and executed. Communication time of each run was recorded.

For a parallel program and for a specific topology of the system graph, let

$$P(s) = \frac{Total\_communication\_time\_under\_a\_random\_mapping}{Total\_communication\_time\_under\_the\_optimal\_mapping}$$

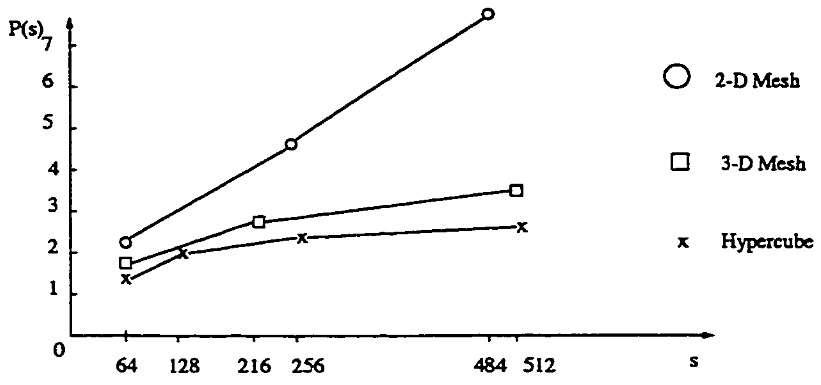
be a function of s where s is the size of the task graph (or the system graph). $P(s)$ denotes the increase in communication time due to link contention caused by random mappings.

The results are shown in Figure 56. We observe that as the size of the task graph increases, the ratio $P(s)$ goes up accordingly as we would expect. We also see that, with the same system size s, $P(s)$ decreases as the system graph changes the topology in the order from line, 2-D mesh, 3-D mesh to hypercube. The reason is that the above order is the increasing order of network connectivity. As the degree of network connectivity is augmented, the degree of contention incurred by random mappings tends to be reduced.

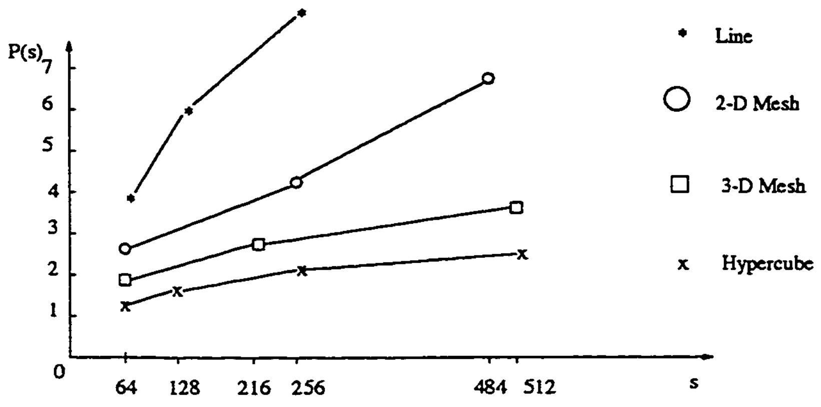
To obtain the results shown in Figure 56, the parameters of the simulated wormhole-routed network were given the following values:

- network topology and size: as shown in Figure 56
- number of lanes per link = 4
- bidirectional links and bidirectional lanes
- buffer size = 1 flit for networks of 256 nodes or less, and 2 flits for networks of more than 256 nodes
- packet size = 8 bytes
- flit size = 8 bits
- flit latency = 1 time unit
- message startup overhead = 10 time units
- packet startup overhead = 10 time units
- header overhead = 5 time units

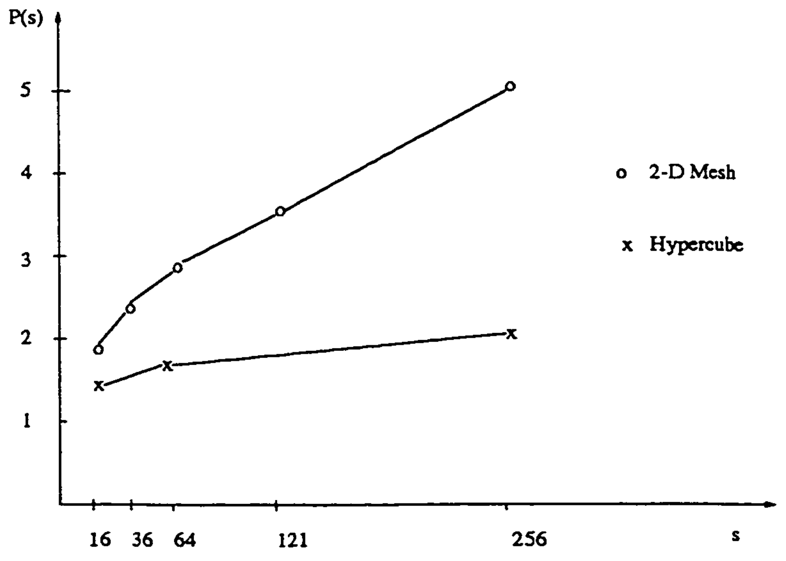
We carried out the same experiment using different sets of parameters. All the outcomes are consistent with the above observations.



Exchange Sort - The task graph is a line of size s



N-Body Program - The task graph is a ring of size s



Matrix Multiplication - The task graph is a 2-D torus of size s

Figure 56: Experimental Results

Chapter 7

Conclusion and Future Work

As part of the CPPE, the CPSS aims at providing flexible and efficient software tools for developing parallel applications and optimizing their performance. It allows users to evaluate impacts of system and software factors on performance of parallel applications. The main objective is to produce efficient parallel programs by locating and eliminating performance bottlenecks in the programs.

The CPSS provides accurate information about the timing and behavior of parallel applications and the underlying simulated architecture. Users are given unprecedented flexibility to adjust physical architectures and system parameters at run time. As a result, the CPSS offers programmers a development environment superior to those available on real multiprocessors. It also outperforms existing simulators in terms of accuracy and flexibility.

The CPSS makes the CPPE an excellent environment for developing and fine-tuning parallel programs. This is due to several advantageous features of the CPSS:

- Accuracy: The CPSS employs the functional simulation technique which offers the most accurate results among the existing simulation techniques. In addition, configurable parameters enables the user to accurately simulate a particular multicomputer system by simply setting the values of system parameters to those belonging to the architecture to be simulated.

- Flexibility: The CPSS can simulate a wide range of multicomputer topologies and sizes. It also supports a large set of configurable parameters which permit users to fine-tune their applications and simulate various multicomputer systems. Moreover, the same virtual-architecture program can be mapped to different physical architectures at run time. The flexibility offered by the CPSS is unique among existing simulators.

- Performance: The simulation is fast because low levels of details are selectively left out to retain essential characteristics of the target processors and network. The entire simulation system, including the application program, is run by a single process, resulting in no host context switching at all.

- Repeatability: The CPSS provides repeatability which is essential for implementing a stable and reliable debugging environment. The CPSS also supports multiple executions of a non-deterministic application. Multiple executions are equally useful for testing the robustness of a deterministic application.

- Correctness and performance debugging tools: The CPSS provides a rich set of correctness and performance debugging tools to facilitate users' code development. Performance statistics at various levels of details are also available to support algorithmic and architectural performance evaluation and tuning.

- User-friendliness and portability: The parallel programming language used in the CPPE is based on the popular language C. Design concepts of the CPSS user interface and debugging tools are borrowed from sequential programming environments. Currently, the simulator can work on UNIX workstations and PCs.

- Expandability: The design and implementation of the simulator are modular and decoupled. Future changes and enhancements to the simulator would be quick and easy.

The CPSS plays a valuable role at all levels of the development of parallel algorithms and applications. It supports testing and debugging, as well as algorithmic and architectural performance evaluation and tuning. It plays an equally vital role in the design and analysis of multicomputer systems, from architectures to network operations.

The simulator is also a powerful tool for parallel research in general. In fact, the performance of the optimal mapping functions for wormhole-routed networks presented in Chapter 6 were validated using the CPSS. Furthermore, the CPSS can be used as a teaching tool for users who wish to learn parallel programming.

The possibilities of expanding the CPSS are numerous. Other routing techniques besides wormhole routing (e.g packet switching, circuit switching) could be supported in the CPSS. This could be done easily due to the modular and decoupled design and implementation of the CPSS. Optimal program mappings could be developed and added to the CPSS mapping library to support efficient communication. The built-in functions library could be extended with more services (e.g. barriers, semaphores) to facilitate users' programming. Currently, a graphical user interface is being developed for the CPSS to enhance user-friendliness.

Appendix A

Parallel Features of the CPC Language

CPC (Concordia Parallel C) language is based on the popular programming language C and enhanced with new features to support parallel programming. The CPC language supports both shared-memory and message-passing programming paradigms. However the scope of this thesis does not cover shared-memory programming paradigm. This appendix will describe only the parallel features supported for message-passing programming style. Parallel features of CPC support the creation of parallel processes, the definition of parallel architectures, process communications through channel variables and mapping of parallel processes to virtual processors. CPC preserves existing sequential features of the C language.

A.1 Creation of Parallel Processes

When a CPC program begins its execution, the simulator creates the very first process, process 0, and assigns it to virtual processor 0 (virtual processor ID = 0) for running. Process 0 will call function *main*(), if any, to start the execution of the parallel program. Note that global variables are considered to belong to process 0 and thus reside on processor 0. Process 0 can then spawn other processes which may then create child processes of their own. This is how parallel activity is initiated in a program: an existing process that is already running on a processor executes a "process creation" statement. A process can create child processes using either *fork* statement or *forall* statement.

A.1.1 *fork* Statement

Process Creation

A *fork* statement followed by any expression will create a new process which will evaluate the expression and run in parallel with the parent. The parent will continue execution right after the creation of the child without waiting for the child to terminate. The child and the parent will then be running in parallel. A simple example is given below.

```
...
n++;
fork printf("Hello world!");
m = n;
...
```

In the above example, after incrementing n, the parent process will spawn a child process which will execute the *printf* statement. While the child is still running, the parent executes the assignment to variable m and runs in parallel with the child.

Syntax

The general syntax of *fork* is as follows:

```
fork <expression>;
```

where `<expression>` can be any CPC valid expression. A new child is spawned, which will evaluate the `<expression>` on a different virtual processor or on the same virtual processor (section A.3 will discuss the mapping of new processes to virtual processors). The parent will continue execution immediately without waiting for the child in any way.

The *fork* statement may precede any CPC expression, causing that whole expression to be executed as a parallel process. Specifically, *fork* may be used in front of a single statement, a block statement, a function call or another *fork* statement as in the following examples.

```
/* The expression is a single statement */
fork printf("Hello world!");
fork while(1)
  {
    int x, y, z;
    ...
    exit;
  }

/* The expression is a block statement */
fork {
  int i, j, k, max;
  ...
  max = FindMax(i, j, k);
  ...
}

/* The expression is a function call  */
fork QuickSort(A);

/* The expression is another fork statement */
fork fork InsertionSort(A);
```

Process Termination

After creating a child using *fork* statement, the parent continues execution immediately without waiting for the child to terminate. Consider the following example:

```
...
for (i = 0, i < n; i++)
   fork Square(i);
...
```

Each iteration of the above *for* loop creates a new process whose code is function *Square*(). As soon as each new child is created, the next loop iteration will proceed immediately. After all n children are created, the *for* loop will end, and execution of the parent process will continue immediately at the statement following the *for* loop. So the parent will be running in parallel with all of its n children.

Although the parent process does continue with its execution while its *fork* children are still running, the parent is not permitted to terminate until all its children have finished. If the parent reaches the end of its code while one or more of its children are still running, the parent will be suspended until all the children terminate. Only then will the parent be allowed to finish. This implementation prevents a premature termination by process 0 while some of its children are still running.

join Statement

There are cases where it is desirable for a parent process to wait at some point for the termination of one or some or all of its children.

join statement is introduced to be used with *fork* in order to delay a parent process until a desired number of its *fork* children have terminated. If the parent has only one *fork* child which is running, the execution of a *join* by the parent will force the parent to wait for the child to terminate. If the child terminated before *join* is executed, the *join* will have no effect on the parent when being executed. An example of the use of *join* is given below.

```
...
fork BubleSort(A);  /* the child runs BubbleSort() */
QuickSort(B);      /* the parent runs QuickSort() */
join;     /* wait for the child to terminate */
for (i = 0; i < N; i++)
   C[i] = A[i] + B[i];
...
```

In this example, the parent executes *join* to wait for the child to terminate. In doing so, the parent ensures that array A is completely sorted before being added to array B which is sorted in parallel with A by the parent.

If a parent has several *fork* children, any *join* in the parent can be matched with any of these children. When executing a *join*, the parent does not specify a particular child; any child who happens to terminate at that time will satisfy the *join*. A *join* is satisfied by one and only one *fork* child's termination. Thus if the parent has multiple *fork* children, it may execute multiple *join* statement to wait for some or all children to finish. Following is an example.

```
...
for (i = 0; i < N; i++)
   TakeAbs(&A[i]);
for (i = 0; i < N; i++)
   join;
...
```

In the above example, the parent creates N processes and then waits for all of them to finish before going further. Without the second *for* loop with *join*, the parent would just proceed, executing in parallel with the children. However, once the parent reaches the end of its code, it would not terminate until all children had terminated.

Note that if the parent process mistakenly executes more *join* statements than it has children, the parent will be blocked forever, resulting in a logic deadlock in the program.

A.1.2 *forall* Statement

Process Creation

forall statement is a parallel form of a normal *for* loop. Each iteration of a *forall* statement creates a child process which will run in parallel with other children created by the same *forall*. In the following example, each process spawned by process 0 calculates the absolute value of the integer assigned to it.

```
main()
{
  int i;
  int a[MAXSIZE];
  ...
  forall i (0; 9; )   /* create 10 processes, for i from 0 to 9*/
    TakeAbs(&(a[i])); /* each process executes TakeAbs() once */
  ...
}
```

The above *forall* statement creates 10 copies of the enclosed assignment statement and makes each one a separate parallel process with its own unique value of the variable *i*. Each of these 10 child processes executes function *TakeAbs*() and may be running on a different processor, all in parallel. Each child process has its own memory space and does not have access to variables belonging to its parent or other processes (except the *forall* index and channel variables that the parent gives up to the new child, as will be described later). As each iteration of the *forall* loop begins, the index *i* is automatically incremented, just as with a normal *for* loop.

After finishing the creation of 10 processes, the parent process suspends its execution, goes to sleep and waits until all of its children terminate. Only then will the parent continue its execution with the statement following the *forall* statement. This is one of the differences between *forall* statement and *fork* statement.

Process Grouping

Process creation incurs overheads (software overheads on the parent processor, birth message send/receive, startup overheads for the new child). So does process termination (software overheads on the child processor, death message send/receive, software overheads on the parent processor). If the process grain is too fine, the overheads may outweigh the speedup gained by parallel processing. In the above example, the time to create and terminate one process is much bigger than the time to execute function TakeAbs which returns the absolute value of an integer. The process grain should be reduced. This is accomplished using *grouping* option which groups together a certain index values in each process. This is illustrated in the following modified version of the above example.

```
main()
{
   int i;
   int a[MAXSIZE];
   ...
   forall i (0; 9; 5)  /* 2 processes are created */
```

```
        TakeAbs(&(a[i])); /* each process executes TakeAbs() 5 times */
      ...
}
```

In this version, only 2 processes are created, each iterating through 5 indices. Process 1 iterates through the indices 0 to 4; process 2 iterates through the indices 5 to 9. The *forall* loop of 10 indices are cut into two parallel normal *for* loops, each iterating through 5 indices. Process creation/termination overheads are reduced from 10 to 2 processes. The grouping size should be chosen so as to balance the program speedup and process creation/termination overheads.

Syntax

Following is the general syntax of the *forall* statement.

```
forall <index_variable>
       (<from_bound>; <to_bound>; {<group_size>})
       <expression>;
```

<from_bound>, <to_bound> and <group_size> can be any integer-valued expressions. If (from_bound > to_bound) then no new child will be created. Expression <group_size> is an optional entity. If <group_size> is omitted, the default group size is 1. If <group_size> does not evenly divide the number of index values in the specified range, then the last child process will have less than <group_size> index values. The code to be executed by the new *forall* children is denoted by <expression>, which can be any valid CPC expression. Examples of <expression> are shown below.

```
/* The expression is a single statement */
forall i (101; 200; 10)
  printf("Hello world!");

/* The expression is a block statement */
forall j (1; 10; )
  {
    char a, b, c, min;
    ...
    min = FindMin(a, b, c);
    ...
  }

/* The expression is a function call */
forall i (0; 9; 4)
  TakeAbs(&(a[i]));

/* The expression is a fork or forall statement */
forall i (1; 10; 5)
  fork PrintResults();

forall i (1; 10; 5)     /* nested forall loops */
  forall k (i; i+9; )
    GetInputs();
```

Nested *forall* Loops

forall statements may be nested to offer greater parallelism. In the following example, two two-dimensional arrays are added using nested *forall* statements.

```
...
typedef twoDarray float[10][5];
...
main()
{
  int i, j;
  twoDarray a, b, c;
  ...
  forall i (1; 10; )     /* 10 processes are created */
    forall j (1; 5; )    /* 50 more processes are created */
      Sum(a[i,j], b[i,j], &c[i,j]);
                      /* each process executes Sum() once */
  ...
}
```

In this example, the outer *forall* incurs the creation of 10 processes, one for each value of *i*. Each of these child processes will consist of an instance of the inner *forall* loop, with the appropriate value of *i*. When each member of this first generation is executed, it will then spawn 5 more processes, one for each index *j*. Thus a total of 50 processes are created in the second generation. The parent process will then have 10 children and 50 grandchildren. So instead of only one physical processor creating all 60 processes, there will be 10 physical processors all creating new children in parallel.

Scope of *forall* Indices

Although children of a process do not have access to variables belonging to the the parent process, the *forall* index is an exception. The child processes can reference the *forall* index as if the loop were a normal *for* loop. Following is an example.

```
main()
{
  int i, k;
  ...
  forall i (0; 9; )
  {
    ...
    printf("Process %d\n", i);       /* this access to i is allowed */
    printf("Value of k = %d\n", k);  /* access to k is not allowed */
         /* will generate run-time error: reference to a non-local */
    ...
  }
  ...
}
```

In this example, the *forall* index *i* is defined as a local to function *main*() which creates 10 child processes. The body of the *forall* loop is the code of every child process. The child processes are allowed to read variable *i* in their code (the first *printf* statement). However they are not permitted to access variable *k*, which is also a local to *main*(). Any attempt from a child to run the second *printf* statement will generate a run-time error: reference to a non-local.

Although the child processes are allowed to access the *forall* index, there is a restriction: references to the *forall* index must be "read" only. In other words, each child process can see only a unique value of the index. Once a process is created and assigned its unique value of the *forall* index, this value cannot be changed within the process. Any attempt to alter the value of this index in an assignment statement will result in a compiler error. Similarly, inside the *forall* body, the *forall* index cannot be passed by reference to a function, or used as the target of an I/O read operation (e.g. *scanf*). The *forall* index may, however, be used in any context that does not change its value. It can be used, for example, in a CPC expression, to index an array, or be passed by value to a function.

Process Termination

A process terminates when it reaches the end of its code. Processes of the same parent may not terminate at the same time. This is due to slight variations in processor speeds, processor loads, or other environmental influences. In any case, the parent process executing the *forall* will always wait for all the child processes to terminate before executing the statement that follows the *forall*.

Consider the general case of the following *forall* statement that creates n children.

```
forall i (0; n-1; )
  <expression>
```

From the parent's perspective, this *forall* results in the following actions:

- Create the n child processes from `<expression>`.
- Wait until all n children have terminated.
- Continue with the statement following the *forall*.

The whole *forall* construct itself is considered as a single statement in the parent process. The execution of that *forall* statement begins by creating the child processes. Then the execution of the parent is temporarily suspended while the children are running on their respective processors. When all the children have terminated, the parent's execution is resumed at the statement that follows the *forall*. This situation is in contrast to a *fork* statement, which allows the parent to proceed right after creating a *fork* child, and run in parallel with the new child.

A.2 Parallel Architecture Definition

The CPC language allows users to specify the virtual architecture in the CPC program. The virtual architecture will then be mapped to a physical architecture at run-time. The physical architecture can be the same as or different from the virtual architecture.

As the CPC language supports both shared-memory and message-passing programming paradigms, an architecture declaration is needed to identify a message-passing program. If the architecture declaration is absent in a program, the program is treated as a shared-memory program. The virtual architecture is specified with the keyword `architecture` at the beginning of the program as in the following example. The architecture of a multicomputer system is defined by the topology and the size of the system.

```
#include<stdio.h>
#define Dim1Size 10
#define Dim2Size 20
architecture mesh my_2D_mesh[Dim1Size][Dim2Size];
...
main()
{
   ...
}
```

The syntax of architecture declaration is as follows:

```
architecture <topology> <name_and_size>
```

where `<topology>` is one of the following topologies: *shared*, *line*, *ring*, *mesh*, *torus*, *hypercube*, and *fullconnect*. *shared* topology means that the program is intended for execution on a shared-memory multiprocessor. In a *fullconnect* topology, each processor is connected to every other processor. `<name_and_size>` is the name and the size of the architecture in the form of an array declaration, as illustrated in the following examples.

```
architecture shared S[100];     /*shared-memory with 100 processors*/
architecture fullconnect F[25]; /*fullconnect with 25 processors*/
architecture line L[10];        /*line with 10 processors*/
architecture ring R[20];        /*ring with 20 processors*/
architecture hypercube H[5];    /*hypercube with 2^5 = 32 processors*/
```

If the topology is *shared*, *line*, *ring* or *fullconnect*, the size of the architecture is the total number of processors. If the topology is *hypercube*, the size of the architecture is the number of dimensions of the hypercube.

For a mesh (or torus), the size of the architecture is defined by the size of each dimension of the mesh (or torus). The minimum number of dimensions of a mesh or torus is 2. Examples are given below.

```
architecture mesh twoDmesh[5][7];            /* 5x7 mesh */
architecture mesh mymesh[3][5][10][8][12];   /* 3x5x10x8x12 mesh */
architecture torus threeDtorus[10][10][10]; /* square 10x10x10 torus*/
```

The above syntax can be rewritten as:

```
architecture line|ring|hypercube|fullconnect|shared <name>[<size>];
architecture mesh|torus <name>[dim1][dim2]{[dim_n]};
```

{`[dim_n]`} here indicates zero or more repetitions of the dimension size. The name of the architecture is required to support multi-phase parallel programs in which different phases of a program may be running on different architectures. The architecture name specifies the architecture to be used for a particular phase. However it is not within the scope of this thesis to describe characteristics and programming rules of multi-phase parallel programs.

A.3 Mapping Processes to Virtual Processors

Users are allowed to map parallel processes to the processors of the virtual architecture to optimize program performance by reducing communication latency. Communicating processes should be mapped to processors sitting close to each other.

Referring back to the syntax of *forall* and *fork* statements, we see that each primitive is ended by an `<expression>` which will be compiled into the parallel code to be executed by the new child. The newly created child can be mapped to a particular virtual processor by preceding the **<expression>** with the absolute ID of that processor. The processor ID can be specified using any valid CPC expression. The value of the expression will be the absolute ID of the virtual processor on which the new child will run. (Please refer to section 4.1 for the conventional conversion between Cartesian IDs and absolute IDs of processors used by the CPC language and CPSS simulator.) Following are two examples of process-to-virtual-processor mapping used with *fork* and *forall* respectively.

```
for (i = 0; i < n; i++)
   fork (i; ) Square(i);


forall k (1; 10; )
   (k-1; ) Cube(k);
```

Thus the syntax of *forall* and *fork* can now be extended as follows:

```
fork {(processor_ID;)} <expression>;


forall <index_variable>
       (<from_bound>; <to_bound>; {<group_size>})
       {(processor_ID; )} <expression>;
```

As indicated in the revised syntax, process-to-virtual-processor mapping is optional. If the user does not specify the virtual processor ID for a new child, at run time, the CPSS will map the child to a default physical processor. The mapping objective is to balance the load among physical processors.

A.4 Process Communication via Channel Variables

Channel variables abstract message sends and receives among processes. A process p can communicate with another process q by "sending" a message through a channel variable. Process q will "receive" the message from the same channel variable.

A message send is abstracted by a write to a channel variable. A message receive is represented by a read from the channel variable. The message forwarding and routing are done by the underlying network simulator, and completely transparent to the programmer. Conceptually, a channel acts like a first-in-first-out queues of values (messages) of the same data type. As values are written to the channel, they are saved in a queue until they are read by some other process. The capacity of the queue buffer is assumed to be unlimited.

A.4.1 Declarations of Channel Variables

A channel variable is declared using the following syntax:

```
channel <component_type> <channel_name>;
```

where `channel` is the reserved word of the CPC language to declare a channel variable. `<component_type>` must be a valid type in the C language. In particular, the component type may be *int*, *float*, *char*, *enum*, array, and structure. The component type is the data type of messages written to or read from the channel. For instance, if the component type is *int*, every message stored in the channel is of type integer. We can also have channel of pointer, and pointer to a channel type. `<channel_name>` is the name of the channel variable. Following are examples of declaring channel variables:

```
typedef enum {Red, Green, Blue} Colors;
typedef char[10] arrayChar;
typedef struct
{ arrayChar name;
  float mark;
} structStudent;

channel int ci;
channel float cf;
channel char cc;
channel Colors ce;
channel arrayChar CA;
channel structStudent CS;
```

A "channel of channel" is not permitted by the rule that the component type must be a valid type in the C language because channel is not a valid type in C.

Channel types may appear at any level of a structured type. For instance, we can have an array of channels. Similarly, one or more fields of a structure may be of type channel. Examples are given below.

```
channel int arrayChan[10];  /*array of 10 channels*/
                            /*each channel is a list of integer values*/
typedef struct
{ int processID;
  channel int mailbox;  /*structure field is of type channel*/
} structProcess;
```

However, nested channels are not allowed: a channel may not contain any channels. For example, we cannot have a channel of structure where one field of the structure is another channel.

We can also define a channel type using the following syntax:

```
typedef channel <component_type> <channel_type_name>;
```

<component_type> must follow the rules stated above, which are:

- The component type must be a valid type in the C language.
- A "channel of channel" is not allowed, and neither are nested channels.

The <channel_type_name> is the name of the new channel type. Following are examples of channel type definitions:

```
typedef channel int chanInt;  /*channel of integer*/
typedef char[20] arrChar;
typedef channel arrChar chanArrChar;
      /*channel of array; the array is 20 characters long*/
chanInt myChanInt;          /*variable of defined type chanInt*/
chanArrChar myChanArrChar; /*variable of defined type chanArrChar*/
```

To allow channel variables to more closely reflect the properties of communication links, the "receiving" end of each channel variable is directly connected to a specific process. The process connected to the "receiving" end of a channel variable is called the *owner* of the channel variable. In the CPC language, only the owner of a channel variable is allowed to read from that channel variable. Each channel variable has one and only one owner (reader).

There are two kinds of owners: *original owners* and *delegated owner*. The original owner of global channel variables is process 0. The original owner of a channel variable local to a function is the process currently executing a copy of the function code which contains that channel variable.

An original owner of a channel variable c can give up the ownership of c to a child it creates, provided that the child can access c according to C lexical scope rules. The child will become the new owner of the channel variable, and only it can read from c (the parent is no longer allowed to read from c; however it can write to c). The child is called the delegated owner of c. The next subsection will describe how a parent process assigns channel variables to its children.

In the CPC language, channel variables are usually declared as global variables. We may have a channel variable c that is local to a function. However this channel would be useless because no processes other than the original owner of c can access c (due to the lexical scope rule of the C language). Thus the original owner cannot use c to communicate with the other processes. That is why channel variables should be declared as global variables so that all processes can see these channel variables.

A.4.2 Binding Channel Variables to New Processes

The following channel declarations will be used for examples in this section, assuming that the channels are global variables.

```
channel int   CI, CJ, CK;          /*channel of integer*/
channel float arrayCF[10];         /*array of 10 channels*/
channel int   arrayCI[20];         /*array of 20 channels*/
channel char  arr3Dchan[5][4][8]; /*3D array of (5x4x8=)160 channels*/
typedef struct
{ int ID;
  float mark;
} StudentStruct
channel StudentStruct myRecord;
```

One or more channels can be assigned to a new process by preceding the new child's code (the `<expression>`) with the names of the channels to be assigned. The syntax of *fork* and *forall* statements presented earlier can thus be extended as follows:

```
fork {({<processor_ID>}; {<channel_list>})} <expression>;

forall <index_variable>
       (<from_bound>; <to_bound>; {<group_size>})
       {(processor_ID; {<channel_list>})} <expression>;
```

The `channel_list` is a list of channel references separated by commas. In this context, a channel reference is defined as being a member of one of the following categories:

1. a channel variable

2. an array of channels

Examples of categories 1 are channels of base type (*int*, *float*, *char*, and *enum*), channel of composite type (array, structure), or an individual element of an array of channels, in which case any valid C expression may also be used to identify subscripts of the array element to be assigned. Examples are given below.

```
main()
{
  int i;
  ...
  fork (1; CI) ChildCode();      /*assign channel CI to new child*/
  fork (4; arrayCF[4]) ChildCode();  /*assign channel arrayCF[4]*/
  ...
  /* Input i, then spawn a new process */
  fork (i; arrayCF[i]) ChildCode();  /*assign channel arrayCF[i]*/
  ...
}
```

A channel reference can be an array of channels to facilitate the binding of many channels of the same component type to a new process. For example, instead of binding 20 channels (of the same component type) to a process, we could declare an array of 20 channels, and then bind that array to the process. Any channel in the array may then be used by that process to receive messages. The array of channels can be an entire array, or one or more dimensions of a multi-dimensional array (e.g. one row of a 2D array, one plan of a 3D array). Following are some examples:

```
main()
{
  ...
  fork ( ; arrayCI) BigChildCode();          /*assign 20 channels*/
  fork ( ; arr3Dchan[1]) BigChildCode();
                                     /*assign 32 channels (1 plane)*/
  fork ( ; arr3Dchan[1][0]) BigChildCode();
                                        /*assign 8 channels (1 row)*/
  ...
}
```

A.4.3 Read and Write on Channel Variables

Channels are written by using their name on the left side of an assignment statement, and read by using their name on the right side of an assignment statement. A channel may be written by many writers but read only by one reader, namely the owner of the channel.

Any process may write values to any channel, provided that the channel variable is accessible by the process according to C lexical scope rules. However, each process may read values only from its own assigned channels.

If the channel is empty, the reader is suspended until some other process writes a value into the channel. However, writer process will never be suspended; channels are supposed to have unlimited capacity and can hold any number of values. Channel writes are thus non-blocking.

Channel Write

Examples of channel writes are shown below. Note that it is not permitted to use a subscript with a *channel of array.* The only operations that can be performed with a *channel of array* is reading or writing a whole array from the channel. It is not allowed to read or write one element in the array. The rules are similar for a *channel of structure*: one may not read or write a single field of the structure.

```
writer()
{
  int i;
  StudentStruct tempRecord;
  ...
  /*Write to channels whose component type is a basic type*/
  CI = 0;
  CI = i + 1;
  arrayCF[1] = 3.1416;
  ...
  /*Write to a channel whose component type is a composite type*/
  tempRecord.ID = 12345;
  myRecord = tempRecord;  /*channel write*/
  ...
}
```

Channel Read

Any channel variable names can be part of an expression on the right side of an assignment statement, as in the following examples:

```
reader()
{
  int i;
  float f;
  StudentStruct bufferRecord;
  ...
  /*Assume that this reader owns the channels used below*/
  ...
  /* Read from channels whose component type is a basic type*/
  i = CJ;
  f = 2 * arrayCF[1] / 5;
  ...
  /* Read from a channel whose component type is a basic type*/
  bufferRecord = myRecord;  /*channel read*/
  bufferRecord.mark++;
  ...
}
```

As stated earlier, for a *channel of structure*, it is not permitted to read a single field of the structure. In the above example, to read field `mark`, one must first read the whole structure from the front of channel `myRecord` into an ordinary structure variable such as `bufferRecord`, and then use the expression `bufferRecord.mark`. The same rules apply to a *channel of array*.

Channel variables may be used in any expression in the program, provided that the component type matches the context in the expression. The general rule is that wherever an ordinary variable of a given type may be used in an expression, a channel variable with that same component type may be used. For example, channel variables may be used as array indices or in Boolean expressions, as in the following examples:

```
anotherReader()
{
  int i, j;
  ...
  /*Assume that this reader owns the channels used below*/
  if (CK) i++;
  else    i = 0;
  ...
  if (arrayCF[5] > arrayCF[6] * 2) i = j;
  ...
}
```

An exception to the above rule is that channel variables are not allowed in I/O statements such as *printf*, and *scanf*.

Note that each time a channel is read, it produces a different value. This is because values are queued inside the channel during writing, and removed during reading. Let CI be a channel of integer. The assignment (n = CI + CI) is not equivalent to (n = CI * 2).

Channel Empty Test

If the channel is empty, the reader is suspended until some other process writes a value into the channel. To avoid this suspension when the channel is empty, the reader can test to determine if the channel currently contains any values. This is achieved by using a boolean-values expression containing the name of the channel variable followed by a question mark, as in the following example for channel *ch*:

```
if (ch?)
  n = ch;  /*read the channel*/
else
  printf("Channel currently empty");
```

The expression "ch?" will evaluate to 1 (true) if channel *ch* currently contains any values and 0 (false) if the channel is empty. The syntax for channel empty test is as follows:

```
<channel_name>?
```

Appendix B

CPSS User's Manual

This appendix serves as a user's manual for using the CPSS interactive system. It begins with general instructions on using the CPSS. Then each command supported in the CPSS is described in details.

B.1 Getting Started

The program is first compiled by the CPCC. To compile a program, at the system prompt, type

```
> cpcc <file_name>
```

where `<file_name>` is the file name of the CPC program to be executed. The file must have a `.c` extension. The file is called *source file*, and the CPC program is called *source program.*

The CPCC will produces a vCode file with the same name as the source file and with extension `.cod`. For instance, if the source file is `ranksort.c`, the vCode file is `ranksort.cod`. The CPSS will execute the vCode instructions contained in the vCode file, and refer to the source file whenever required (e.g. setting breakpoint at source line n).

To enter the CPSS, at the system prompt, type

```
> cpss
```

Inside the CPSS, vCode files can be loaded for execution, one at a time using LOAD command. The vCode of the loaded file is stored in the *code buffer* of the simulator. If a file name is specified with cpss command as in the following example:

```
> cpss ranksort
```

that file is loaded automatically into the code buffer when the user enters the CPSS, and ready for execution.

B.2 An Overview of CPSS Commands

Following is a list of interactive commands supported by the CPSS. These commands are issues directly in the CPSS in response to the CPSS prompt.

- LOAD *program* - Loads the program whose file name is *program* for execution.
- RUN - Runs the whole program from the beginning.
- QUIT - Quits the CPSS simulator.
- LIST m:n - Lists program source lines m through n on the screen.
- ARCH *topology size* - Sets the physical architecture.
- MAP - Gets the virtual-to-physical-architecture mapping from the user.
- PARAMETER DIS/CHA - Displays/changes system parameters.
- BREAK n - Sets a breakpoint at program line n. Program execution will be suspended whenever any process attempts to execute this line of the program.
- CLEAR BREAK n - Clears the breakpoint from program line n.
- CONT - Continues execution of the program after a breakpoint has been encountered.
- STEP n - Continues execution for n lines in the current process, then suspends program execution again.

- WRITE *p var* - Displays the current value of variable *var* belonging to process *p*.
- TRACE *p var* - Makes variable *var* belonging to process *p* a trace variable. Whenever any process attempts to read or write a trace variable, the program execution is suspended.
- CLEAR TRACE *m* - Clears the trace from memory location *m*.
- DISPLAY - Displays list of breakpoints, trace variables, and the alarm.
- ALARM *t* - Sets an alarm to go off after *t* time units from the beginning of program execution. Program execution is suspended when the alarm goes off.
- STATUS *p* - Displays information about the current status of process *p*.
- TIME - Gives the elapsed time since the start of program and since the most recent breakpoint.
- UTILIZATION *p* - Gives the utilization percentage for physical processor *p*.
- VARYSPEED ON (OFF) - Creates randomly chosen variations in the speed of processors, to help the user determine if the program has timing-dependent bugs (section 3.5.4).

B.3 Basic Commands

To load a new program into the code buffer for execution, use the LOAD command followed by the name of the vCode file (with or without .cod extension). Examples are:

```
>> LOAD ranksort
>> LOAD matrix_mult.cod
```

When this command is issued, if there is another CPC program currently residing in the code buffer, that program is cleared from the buffer. The CPSS will search for the file to be loaded, and fetch the vCode into the code buffer. The CPSS does not need the source file for program execution. It only needs the source file if the user issues commands which reference the source file (e.g. listing source lines).

To run the program, simply type the following:

```
>> RUN
```

Inside the CPSS, the user may at any time get a listing of the whole program (including line numbers) with the command:

```
>> LIST
```

To get a listing of a portion of the program with line numbers, indicate the range "m:n" of line numbers, as in the following example:

```
>> LIST 25:35
```

To quit the CPSS environment at any CPSS prompt, type the following:

```
>> EXIT
```

B.4 Setting the Physical Architecture

The CPSS can run the same CPC program on different physical architectures. To set the physical architecture, type the command:

```
>> ARCH <topology> <size>
```

where `<topology>` is one of the following: *full* (full connect), *line*, *ring*, *hypercube*, *mesh*, and *torus*. If `<topology>` is *full*, *line* or *ring*, `<size>` is the total number of physical processors. If `<topology>` is *hypercube*, `<size>` is the number of dimensions of the hypercube. If `<topology>` is *mesh* or *torus*, `<size>` is a list of integers, each integer being the number of processors along a dimension of the mesh or torus. Thus the length of the list of integers is the number of dimensions of the mesh or torus. Examples are given below.

```
>> ARCH ring 128
>> ARCH hypercube 7
>> ARCH mesh 4 8 4
```

In the above examples, the ring, hypercube and 3D mesh have the same number of processors, which is 128.

Setting the physical architecture must be done at the very beginning of a run. It is not allowed in the middle of program execution (e.g. at a breakpoint).

If no physical architecture is specified for a newly loaded program, the default physical architecture is the same as the virtual architecture indicated in the source program.

B.5 Displaying and Changing System Parameters

To view values of the system parameters, type the following command:

```
>> PARAMETER DIS
```

System information similar to the following will be displayed:

```
Number of lanes/link = 2
Flit size       = 1 bytes
Buffer size     = 1 flits
Packet size     = 4 bytes = 4 flits
Header size     = 2 flits
Packet data     = 2 flits

Startup_cost/message    = 5
Startup_cost/packet     = 2
Non_head_flit_speed/Head_flit_speed     = 4
```

To change the value of a parameter, issue the following command:

```
>> PARAMETER CHA
```

A list of system parameters is then shown for selection. After the user specifies the parameter to be changed, the CPSS prompts for the new value. The CPSS will get the entered value, verify it, and update the parameter if the entered value is valid. The user can use command "PARAMETER DIS" to verify the update.

B.6 Displaying Process Status

When encountering a breakpoint, the user may use the following command to determine the status and current execution point of each active process:

```
>> STATUS
```

An example output is as follows.

```
Process Nbr | Function | Line Nbr | Status | VirProcNbr | PhyProcNbr
=====================================================================
      0           main       46     Ready          0           0
      1       pipeproc       17     Blocked        1           0
      2       pipeproc       17     Blocked        2           0
      3       pipeproc       17     Running        3           0
```

To display the status of a particular process, specify the process ID in the STATUS command, as in the following example:

```
>> STATUS 4
```

If the current number of active processes is large, the user can choose to view only a range of processes as in the following command:

```
>> STATUS 10:20
```

The status of processes with ID from 10 to 20 will be displayed.

B.7 Setting Breakpoints

In the CPSS system, breakpoints are set by referring to program line numbers. A breakpoint may be set on any executable line of the source program with the following command:

```
>> BREAK <line_number>
```

To get a list of the location of all breakpoints, use the following command:

```
>> DISPLAY
```

Individual breakpoints may be removed one at a time as follows:

```
>> CLEAR BREAK <line_number>
```

The program runs until any line of the source program with a breakpoint is hit. The execution will be suspended immediately prior to the execution of the line with the breakpoint. Then the CPSS prompt will appear, and the user may then set or remove breakpoints, examine the values of program variables, or use any of the other debugging commands at this point. To continue execution of the program to the next breakpoint, use the following command:

```
>> CONTINUE
```

B.8 Stepping Through a Process

Each breakpoint is valid for all the processes. When any running process tries to execute a line with a breakpoint, the whole program execution will be suspended, including the execution of all the processes. To focus on the execution of a specific process, the STEP command may be used to follow the execution of that process line by line. Whenever a breakpoint is encountered by a given process, that process automatically becomes the current *stepped process*. All STEP commands refer to only the current stepped process.

The current stepped process may be changed, as in the following example that changes the stepped process to process number 9:

```
>> STEP PROCESS 9
```

To trace the execution of the current stepped process, the STEP command can execute a specified number of lines in the stepped process, and then suspends execution. In the following example, the stepped process will execute five lines of the source program:

```
>> STEP 5
```

Note that blank lines, comment lines, and declarations are not counted by the STEP command. Also note that during a STEP command, breakpoints are ignored. Breakpoints are valid only during the CONTINUE command or the RUN command.

B.9 Writing Variables

In a CPC program, each process has its own individual execution flow and execution environment. A variable is said to be currently in the environment of a process if that process has legal access to that variable. When the program execution is suspended (e.g. due to a breakpoint), the CPSS allows the user to display the value of any variable in the current environment of each process with a command of the following general form:

```
>> WRITE <process_number> <variable_name>
```

The `<process_number>` must be the ID of some active process that appears in the STATUS command list. The `<variable_name>` is located in the environment of that process, and the value of the variable will be displayed.

Let the following variables be currently in the environment of an active process having process ID number 4.

```
int i;
float f;
char c;
enum {Red, Green, Blue} e;
struct
{ int ID;
  int age;
  float salary;
} s;
int A[10];
channel int CI;
```

Examples of using WRITE command are given below.

B.9.1 Variables of Basic Types

Examples of basic types are *int*, *char*, *float*, and *enum* (enumeration). The following commands display the values of basic-type variables of process 4.

```
>> WRITE 4 i

>> WRITE 4 f

>> WRITE 4 c
```

B.9.2 Variables of Type Structure

The WRITE command allows the user to refer to a specific item of a structure as in the following commands:

```
>> WRITE 4 s.ID

>> WRITE 4 s.salary
```

By using the name of a structure, the whole structure will be displayed, including the names and values of all its components as in the following example:

```
>> WRITE 4 s

structure
   ID       2834952
   age      25
   salary   5,578.50
```

B.9.3 Variables of Type Array

In the WRITE command, it is permitted to refer to a specific item of an array as in the following example:

```
>> WRITE 4 A[1]
```

A range of indices in an array may be displayed by requesting to WRITE the whole array as follows:

```
>> WRITE 4 A
```

The CPSS will respond with the prompt message "`Index Range >`". To display the entire array, the user simply presses the Return key. The user may also specify any range of indices as in the following example:

```
>> WRITE 4 A
Index Range > 5:9
```

B.9.4 Variables of Type Channel

A *channel* variable contains a list of values with new values added at the tail and removes from the head. Using the WRITE command, the current contents of any *channel* variable may be displayed as in the following example for a variable *ci* declared as `channel int ci`:

```
>> WRITE ci

ci = 8
   = 0
   = -20
```

The channel contents are written with the head at the top and the tail at the bottom. It may happen that a value is inserted into the list of channel values before it is actually available. Such values are also listed as part of the channel contents, but preceded with a special "**" notation to indicate that they are currently unavailable for reading. Below is an example.

```
>> WRITE ci

ci = 15
   = 3
** = 1
** = 25
```

B.10 Tracing Variables

Besides the BREAK and STEP commands, another command which enables the user to interrupt program execution is the TRACE command. This command puts a trace flag on any variable. Whenever that variable is referenced during subsequent program execution, the program will be suspended as it is for breakpoints. The general form of the TRACE command is as follows:

```
>> TRACE <process_number> <variable_reference>
```

The `<process_number>` plays the same role as in the WRITE command to identify the process and its environment. The `<variable_reference>` must be a fully qualified reference that evaluates to a scalar variable of type *int*, *char*, *float*, *enum* (an *enum* variable is evaluated to an *int*) or a single *channel*. Individual elements of an array or structure may be traced, but one cannot use a single TRACE command for an entire array or structure. With reference to the sample variables declaration in section B.9.2, the following are all valid TRACE commands:

```
>> TRACE 4 i

>> TRACE 4 f

>> TRACE 4 s.ID

>> TRACE 4 A[9]

>> TRACE 4 CI
```

However, the following commands are not accepted by the CPSS, since they refer to the whole array/structure:

```
>> TRACE 4 s
Cannot trace a whole array or structure.

>> TRACE 4 A
Cannot trace a whole array or structure.
```

As with the WRITE command, a TRACE command must refer to variables that already exist in the environment of the specified process. To get a list of the currently active trace variables, use the DISPLAY command, which displays the breakpoint locations and the trace locations as in the following example:

```
>> DISPLAY

Breakpoints on the following lines:
17
29
Trace Variable Name         Memory Location
  internal                        427
  i                               354
```

To remove a trace from a variable, use the following command:

```
>> CLEAR TRACE <memory_location>
```

where `<memory_location>` is obtained from the DISPLAY command (the memory location is the absolute address of the trace variable in the memory pool of the simulator). Below are some examples:

```
>> CLEAR TRACE 427

>> CLEAR TRACE 354
```

B.11 TIME Command

The TIME command can be used whenever program execution is suspended to give the total elapsed time since the beginning of the program and since the last breakpoint (if any). Following is a typical output from the TIME command:

```
>> TIME

Since the beginning:
      Elapsed Time : 390
      Number of Processors Used:  4
      SequentialTime/ParallelTime: 0.98
Since last breakpoint:
      Elapsed Time: 100
      SequentialTime/ParallelTime: 1.02
```

Using the TIME command between breakpoints is useful for focusing attention on the performance of localized segments of the program. For example, the user can set a breakpoint and then after hitting this breakpoint, run the program up to a second breakpoint. The TIME command will then provide information about the performance of the program between the breakpoints.

B.12 Displaying Processor Utilization

The utilization of a given physical processor is defined as the proportion of the time the processor is actually running. Any time the program execution is suspended, the user may issue the UTILIZATION command to obtain a table of the utilization of all processors up to that point in the execution. This command will also give the processor utilization since the last breakpoint: the proportion of time since the last breakpoint that each processor is actually running. Following is a typical display from the UTILIZATION command:

```
>> UTILIZATION


             Utilization Percentage
Processor | Since the beginning | Since last breakpoint
======================================================
    0             70                       65
    1             54                       74
    2             48                       68
    3             51                       72
```

When using the UTILIZATION command, the display will automatically include all processors that have been used since the start of the program. Other processors not listed all have zero utilization. If the list of processors is too long, the user may specify a range of processors to be displayed as follows:

```
>> UTILIZATION 25:40
```

B.13 Setting the Alarm

It is sometimes useful to be able to stop the program execution at a specific time. The ALARM command is supported for this purpose: it will automatically suspend program execution when a certain time is reached. This is similar to setting an alarm on an ordinary watch or clock. The following command sets an alarm to go off at 500 time units:

```
>> ALARM 500
```

The alarm time is always measured from the start of program execution initiated with the RUN command. When the program time reaches 500, the execution will be suspended and the following message will be displayed:

```
>> Time is 500.  Alarm went off.
```

In some cases, the program may actually stop slightly after the specified alarm time. Once set with the ALARM command, the alarm setting will continue to remain valid. The alarm may be reset to a new time if desired. The user can also turn off the alarm entirely with the following command:

```
>> ALARM OFF
```

The DISPLAY command, in addition to listing the breakpoints and trace variables, will also show the alarm setting if it is on.

B.14 VARYSPEED Command

This command is used for testing multiple executions of non-deterministic applications and robustness of deterministic programs, as discussed in section 3.5.4. Race conditions are simulated by varying relative processor speeds.

The VARYSPEED option is turned on using the following command:

```
>> VARYSPEED ON
```

After this command is issued, the program may be executed by using the RUN command as usual. When the RUN command is issued, the user will be prompted for an integer "Random Number Seed." The seed 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, until another RUN command is issued, at which time a new Random Number Seed will be requested to select a new set of random speed factors for the processors. 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 off the VARYSPEED option, use the following command:

```
>> VARYSPEED OFF
```

Appendix C

CPPE's vCode Instructions

This appendix provides a list of vCode instructions used in the CPPE. The instructions are classified based on their functionality.

C.1 List of vCode Instructions

The set of vCode instructions is categorized into eight groups based on their functionality:

1. Load/store instructions
2. Other memory operations
3. Instructions implementing conditional statements
4. Instructions handling function calls
5. Instructions handling parallel processes and their execution
6. Channel operations
7. Mathematical and logical operations
8. Miscellaneous instructions

Instructions belonged to each group are listed below.

C.1.1 Load/Store

Mnemonic	Description
LDVal	Load value of variable V onto stack
LDAddr	Load address of variable V onto stack
LDIndirect	Load indirectly from variable V
LDReal	Load a real literal onto stack
LDInt	Load an integer literal onto stack
LDSP	Load the stack top pointer onto stack
STORE	Store value into memory location
LoadBlock	Load a block of data onto stack
CopyBlock	Copy a block of data into another
CopyToNewBlock	Allocate a new block and copy data to it

C.1.2 Other Memory Operations

Mnemonic	Description
IndxArray	Array indexing
New	Dynamic allocation for pointer variables
Dispose	Dynamic deallocation for pointer variables
Dereference	Replace pointer on the stack by value
DupTop	Duplicate the stack top
Pop	Pop the top element off the stack
ChecklVarAccess	Check variable access right of the current process

C.1.3 Conditional Statements

Mnemonic	Description
JMP	Unconditional jump
JMPZ	Conditional jump on zero
SwBeg	Begin of *switch* statement
SwTab	Switch table entry

C.1.4 Mathematical and Logical Operations

Mnemonic	Description
ShiftL	Shift integer to left
ShiftR	Shift integer to right
BitOr	Bitwise OR
BitXor	Bitwise XOR
BitAnd	Bitwise AND
BitNot	Bitwise NOT
Equal	Equal test
NotEqual	Not equal test
LT	$<$ test
LE	$<=$ test
GT	$>$ test
GE	$>=$ test
BoolAND	Logical AND
BoolOR	Logical OR
BoolNOT	Logical NOT
NEGATE	Negate value on stack
ADD	Addition
SUB	Subtraction
MUL	Multiplication
DIV	Division
MOD	Modulo
IntToFloat	Convert the integer on the stack top to a float
FloatToInt	Convert the float on the stack top to an integer

C.1.5 Function Calls

Mnemonic	Description
NewFrame	Allocate a new function frame
Call	Call function
ExitFunc	Exit function

C.1.6 Parallel Processes

Mnemonic	Description
BeginParallel	Begin parallel execution of *forall* loop
EndParallel	End of parent's execution in a *forall* loop
NewForkChild	Create a new *fork* child
NewForallChild	Create a new *forall* child
ForkChildEnd	End a *fork* process
ForallChildEnd	End a *forall* process
BeginForallLoop	Start of the *forall* loop body
EndForallLoop	End of the *forall* loop body
SonLDForallIndexVal	Load *forall* index from stack by child
DadLDForallIndexVal	Load *forall* index from stack by parent
TstGrpIncIdx	Test *grouping* index at the end of *forall*
Wakeup	Wake up parent after parameter evaluation
ForkJump	Special jump with *fork*
JOIN	Wait for a *fork* child process to merge
DefaultProc	Default processor for a new process
SwitchOff	Context switching is disabled
SwitchOn	Context switching is enabled
HALT	Process 0 attempts to terminate

C.1.7 Channel Operations

Mnemonic	Description
LDCHwOffset	Load channel variable
LDCHwOffsetInd	Load channel variable indirect
LDCHwAdrOnS	Load channel with address on stack
STChannel	Store value from top of stack into channel
TstCHwAdrOnS	Test channel with address on stack
TstCHwOffset	Test channel variable
TstCHwOffsetInd	Test channel indirect
MVChannVar	Bind a channel variable to a new process

C.1.8 Miscellaneous

Mnemonic	Description
NOP	No operation/execution
BuiltinFunc	Built-in Function

Bibliography

[1] S. Chittor and R. Enbody, "Performance evaluation of mesh-connected wormhole-routed networks for interprocessor communication in multicomputers," *Proceedings of the Supercomputing Conference*, November 1990, pp.647-656

[2] 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

[3] S. Chittor and R. Enbody, "Hypercubes vs. 2D meshes," *Proceedings of the SIAM Fourth Annual Conference on Parallel Processing for Scientific Computing*, 1990, pp.313-318

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

[5] G.C. Fox, et al, *Solving problems on concurrent processors, Vol. I, General techniques and regular problems*, Prentice-Hall, Englwood Cliffs, NJ, 1988

[6] E. Ma and L. Tao, "Embedding among toruses and meshes," *Proceedings of the International Conference on Parallel Processing*, 1987, pp.178-187

[7] E. Ma and L. Tao, "Embedding among meshes and tori," *Journal of Parallel and Distributed Computing*, vol.18, no.1, 1993, pp.44-55.

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

[9] 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

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

[11] A. Borg, R.E. Kessler, and D.W. Wall, "Generation and analysis of very long address traces," *Proceedings of the 17th Annual International Sumposium on Computer Architecture*, 1990, pp.270-279

[12] 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

[13] D. Chaiken, B.H. Lim, and D. Nussbaum, *ASIM User Manual*, ALEWIFE Systems Memo 13, August 1990

[14] D. Culler, et al, "LogP: Towards a realistic model of parallel computation", *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993, pp.1-12

[15] W.J. Dally and C.L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Transactions on Computers*, May 1987, vol.C-36, no.5, pp.547-553

[16] W.J. Dally and P. Song, "Design of a self-timed VLSI multicomputer communication controller," Proceedings of the 1987 International Conference on Computer Design, IEEE CS Press, 1987, pp.230-234

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

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

[19] 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

[20] J.T. Draper and J. Ghosh, "A comprehensive analytic model for wormhole routing in multicomputer systems," *Journal of Parallel and Distributed Computing*, November 1994, vol.23, pp.202-214

[21] S.J. Eggers, et al, "Technique for efficient inline tracing on a shared-memory multiprocessor," *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1990, pp.37-47

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

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

[24] J. Kim and C.R. Das, "Hypercube communication delay with wormhole routing," *IEEE Transactions on Computers*, July 1994, vol.43, pp.806-814

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

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

[27] S.K. Reinhardt, et al, "The wisconsin wind tunnel: virtual prototyping of parallel computers," *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1993, pp.48-60

[28] J. Rexford, et al, "PP-MESS-SIM: A simulator for evaluating multicomputer interconnection networks," *Proceedings of the 28th Annual Simulation Symposium*, 1995, pp.84-93

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

[30] D. Zukowski, et al, "XPOSÉ: A simulator for network development," *Proceedings of the 27th Annual Simulation Symposium*, 1994, pp.59-68

[31] A. Silberschatz, J. Peterson and P. Galvin, *Operating System Comcepts*, 3rd edition, Addison-Wesley, 1991

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

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

[34] 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

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

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

[37] G.A. Geist, et al, *A users' guide to PICL: a portable instrumented communication library*, Oak Ridge, TN, August 1990

[38] X. Lin and L.M. Mi, "Deadlock-free multicast wormhole routing in multicomputer networks," *Proceedings of the 18th International Symposium on Computer Architecture*, IEEE CS Press, 1991, pp.116-125

[39] S. Konstantinidou, "Adaptive, minimal routing in hypercubes," *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI*, MIT Press, 1990, pp.139-153

[40] W.J. Dally and H. Aoki, "Adaptive routing using virtual channels," Technical Report, Massachusetts Institute of Technology, Laboratory of Computer Science, September 1990

[41] D.H. Linder and J.C. Harden, "An adaptive and fault-tolerant wormhole routing strategy for k-ary n-cubes," *IEEE Transactions on Computers*, January 1991, vol.40, no.1, pp.2-12

[42] C.J. Glass and L.M. Ni, "The Turn model for adaptive routing," *Proceedings of the 19th International Symposium on Computer Architecture*, IEEE CS Press, 1992, pp.278-287

[43] Anonymous, "Commercial parallel computer line uses VLSI to cut number-crunching costs," *Computer Systems Equipment Design*, March 1985, pp.9-13

[44] J.P. Hayes, et al, "Architecture of a hypercube supercomputer," *Proceedings of the 1986 International Conference on Parallel Processing*, 1986, pp.653-660

[45] J.L. Gustafson, et al, "Architecture of a homogeneous vector supercomputer," *Proceedings of the 1986 International Conference on Parallel Processing*, 1986, pp.649-652

[46] P. Kermani and L. Kleinrock, "Virtual cut-through: a new communication switching technique," *Computer Networks*, 1979, vol.3, no.4, pp.267-286

[47] G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummnings Publishing Company, 1989

[48] *nCUBE 6400 Processor Manual*, *n*CUBE Company, Beaverton, OR 97006, 1990

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

[50] S. Borkar, et al, "Supporting Systolic and Memory Communication in iWarp," *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990, pp.70-81

[51] M. Homewood, et al, "The IMS T800 Transputer," *IEEE Micro*, 1987, vol.7, no.5, pp.10-26

[52] T.H. Dunigan, "Performance of a second generation hypercube," Technical Report ORNL/TM-10881, Oak Ridge National Lab, November 1988.

[53] Intel. Personal Communication, 1991

[54] T. von Eiken, et al, "Active messages: a mechanism for integrated communication and computation," *Proceedings of the 19th International Symposium on Computer Architecture*, 1992, pp.256-266