Brno University of Technology
Faculty of Information Technology

Parallel Performance Modeling, Prediction and Tuning

Doctoral Thesis

Author: Ing. Jiří Staroba
Supervisor: Prof. Ing. Václav Dvořák, DrSc.

Brno 2004

Abstract

This thesis presents a unified approach to modeling of parallel architectures and algorithms with special emphasis on estimation of obtainable performance. A modeling language and simulator *Transim* is used for this purpose. Although Transim was designed by its authors just as a transputer simulator for prototyping and performance evaluation of message-passing programs, it is applied to simulations of many different types of parallel architectures and programming paradigms, what is far beyond the originally anticipated applications.

The approach is demonstrated on simulations of abstract machine models like PRAM or APRAM as well as commonly used parallel architectures like symmetrical multiprocessors, clusters of workstations and their combinations. Performance tuning of parallel algorithms is undertaken and results of simulations are compared to results obtained on real parallel computers. Presented simulation models also include various synchronization operations found in many parallel algorithms. These models can be used as building blocks of more complex models.

Finally performance tuning of communication algorithms has been undertaken, because of the dramatic impact of these algorithms on the overhead of parallel computing. Communication is an indispensable part of any parallel computation and the results are therefore applicable to a wide class of parallel applications running on distributed machines with irregular network topology.

Keywords: modeling, simulation, parallel performance, performance prediction, performance tuning, parallel computing, parallel architectures, parallel algorithms, genetic algorithms.

Acknowledgements

I would like to thank to Professor Václav Dvořák for supervising my work and his continuous support and helpful advice during last years.

I am also very grateful to my wife Olga for her patience and moral support in my work.

Contents

1 Introduction **1**
1.1 Parallel Architectures . . . 2
1.2 Parallel Programming Models . . . 5
1.3 Performance Factors . . . 6
1.3.1 Granularity . . . 6
1.3.2 Speedup and Its Limitations . . . 6
1.4 State of the Art . . . 7
1.5 Dissertation Goals and Used Methods . . . 10
1.5.1 Transim Description . . . 10
1.6 Overview of the Following Chapters . . . 13

2 Abstract Models of Parallel Systems **15**
2.1 PRAM . . . 15
2.2 Asynchronous PRAM (APRAM) . . . 16
2.3 Bulk Synchronous Parallel (BSP) Model . . . 17
2.4 Simulating PRAM Algorithms with Transim . . . 17
2.4.1 Performance Tuning of a Sorting Algorithm . . . 18
2.4.2 Results of a Simulation . . . 20
2.5 Conclusions . . . 21

3 Bus-based Symmetrical Multiprocessors **22**
3.1 Modeling CCBB Multiprocessors . . . 22
3.1.1 Fairness and Overhead of `ALT` and Arbiters . . . 26
3.2 Models of Synchronization Primitives . . . 27
3.2.1 Locks . . . 27
3.2.2 Barriers . . . 29
3.2.3 Simulation of Synchronization Performance . . . 31
3.3 Parallel Benchmark Program . . . 32
3.3.1 Algorithm Description . . . 32
3.3.2 OpenMP Implementation . . . 34
3.3.3 Simulation Model Parameters . . . 35
3.4 Results And Conclusions . . . 36

4 Clusters of Workstations and SMPs **37**
4.1 Cluster Interconnection Models . . . 37
4.1.1 Omega Network . . . 37
4.1.2 Fat Tree Topology . . . 39

4.1.3 A Crossbar Switch . . . 40
4.1.4 Models of Switching Techniques . . . 41
4.2 SMP Cluster Model . . . 42
4.3 A System of Linear Equations Benchmark . . . 44
4.3.1 Simulation of a Workstation Cluster . . . 44
4.3.2 SMP Cluster Simulation . . . 46
4.4 Conclusions . . . 47

5 Tuning the Performance of Communication Algorithms **49**
5.1 Models of Communications on Irregular Topologies . . . 50
5.2 GAroute Algorithm Description . . . 53
5.2.1 Input . . . 54
5.2.2 The Search for the Shortest Paths . . . 55
5.2.3 The Optimization Algorithm . . . 56
5.2.4 Sequential Performance Tuning . . . 57
5.2.5 Parallel Performance Tuning . . . 58
5.2.6 Experimental Results . . . 61
5.3 Conclusions . . . 63

6 Conclusions and Future Research Directions **64**
6.1 Contributions . . . 64
6.2 Future Research Directions . . . 66

A Transim Language Reference **67**
A.1 Software Description . . . 67
A.1.1 Data Types . . . 67
A.1.2 Code Structuring . . . 67
A.1.3 Communication . . . 68
A.1.4 Predefined Functions and Procedures . . . 68
A.2 Hardware Description . . . 69
A.3 Mapping Software onto Hardware . . . 69

Bibliography **70**

List of Figures **76**

List of Tables **78**

Chapter 1

Introduction

Parallel processing is as old as computers themselves. At the beginning it was motivated purely by unreliability of first computers. The same computations were run on more than one computer to increase probability of obtaining correct results. As reliability of computers increased, this sort of parallelism survived only in mission-critical applications, where very high reliability and availability are desired. In these applications, redundant systems, or their parts, are used to obtain a certain degree of *fault tolerance*.

The concept of parallel computing in the sense in which it will be understood in the remainder of this thesis refers to the use of multiple processors operating together on a single problem, which potentially provides significantly higher performance than using a single processor. The problem at hand has to be split into parts (code, data, or both), each of which is solved by a separate processor. The computing platform, which is referred to as a *parallel computer*, could be a specially designed computer system containing multiple processors or several independent computers interconnected in some way.

Parallel computing has been forced by increasing demand for faster computer systems. The idea is that P computers could provide up to P times the computational power of a single computer, no matter what is the power of a single computer, with the expectation that the problem would be completed in $1/P^{\text{th}}$ of the time or a much larger problem would be solved in the original time. Of course, this is an ideal situation that is rarely achieved in practice. Problems often cannot be divided perfectly into independent parts. Some interaction is necessary between the parts, both for data transfer and for synchronization of computations. However substantial speedup of a solution may be achieved depending upon the problem and the amount of parallelism in the problem.

Even though speed of processors, as well as performance of other computer devices, has raised almost exponentially during the last decades [1] and this tendency can be expected to hold also for some years to come, there is a constant demand of new applications for more computational power than uniprocessor systems can deliver. What also makes parallel computing timeless is that the continual improvements in execution speeds of single processors simply make parallel computers even faster and there will always be problems that cannot be solved in a reasonable amount of time on current computers — not only grand challenge problems, but

also signal and image processing in real time, robotics, knowledge mining and even consumer electronics.

Apart from obtaining the potential for increased speed on an existing problem, the use of multiple computers/processors often allows a larger problem or a more precise solution of a problem to be solved in a reasonable amount of time. For example, computing many physical phenomena involves dividing the problem into discrete solution points. Forecasting the weather involves dividing the space into a three-dimensional grid of solution points. Two- and three-dimensional grids of solution points occur in many other applications. A multiprocessor solution will often allow more solution points to be computed in a given time, and hence a more precise solution. A related factor is also that multiple computers often have more total main memory than a single computer, enabling solution of problems that require larger amounts of main memory to store data structures.

The idea of using parallel computing to improve performance is not new. Articles about parallel computers appeared already in 1950s. For example, Gill writes about parallel programming in 1958 [2] and Holland writes about a "computer capable of executing an arbitrary number of sub-programs simultaneously" in 1959 [3]. Despite its relatively long history, parallel computing is still an active research area with large potential for improvements. At present we are witnessing a rebirth of parallel processing on a single chip [4].

1.1 Parallel Architectures

One of fundamental taxonomies of computer architectures proposed already in 1966 by Flynn [5], but still useful today, is a model of categorizing all computers into four classes according to parallelism at the instruction stream and data stream levels. These categories combine single/multiple data streams and single/multiple instruction streams. From the four possible combinations, the only category, which emerged as the parallel architecture of choice for general-purpose multiprocessors, is MIMD (multiple instruction streams, multiple data streams). This is primarily due to two reasons:

1. MIMDs offer flexibility. With the correct hardware and software support, MIMDs can function as single-user multiprocessors focusing on high performance for one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of these functions.

2. MIMDs can build on the cost-performance advantages of off-the-shelf microprocessors. In fact, nearly all multiprocessors built today use the same microprocessors found in workstations and single-processor servers.

Existing MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictate a memory organization and interconnect strategy. The first group, called *centralized shared-memory architectures*, usually does not have more than a few tens of processors. The second group, which consists of multiprocessors with physically distributed memory, scales to hundreds or thousands processors.

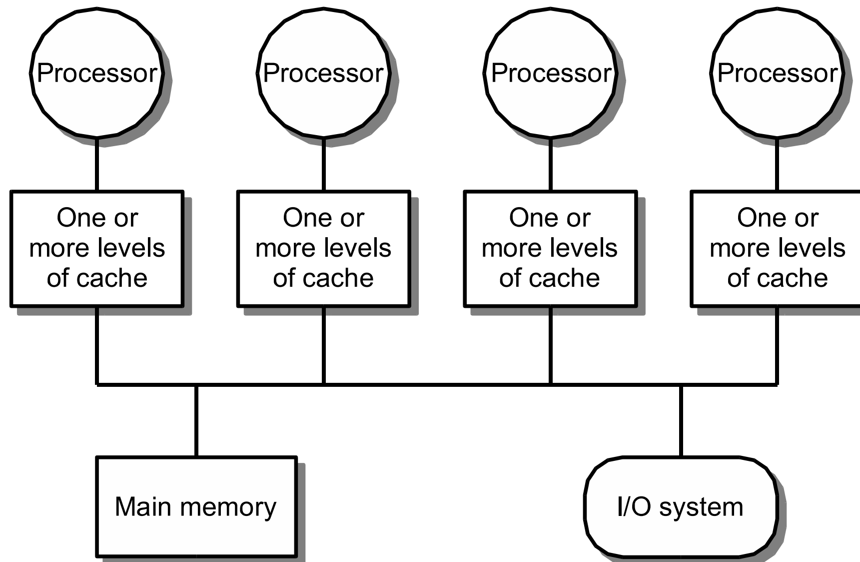


Fig. 1.1: Basic structure of a shared-memory multiprocessor

For multiprocessors with small processor counts, it is possible for the processors to share a single centralized memory and to interconnect the processors and memory by a bus. With large caches, the bus and the single memory, possibly with multiple banks, can satisfy the memory demands of a small number of processors. By replacing a single bus with multiple buses, or even with a switch, a centralized shared-memory design can be scaled to a few tens of processors. Due to a single main memory that has a symmetric relationship to all processors and uniform access time from any processor these multiprocessors are often called *symmetric (shared-memory) multiprocessors* (SMPs), and this style of architecture is sometimes called *uniform memory access* (UMA). Figure 1.1 shows what these multiprocessors look like.

To support a larger processor count, memory in parallel architectures must be distributed among the processors rather than centralized; otherwise the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring excessively long access latency. The larger number of processors raises the need for a high bandwidth interconnect. Both direct interconnection networks (with each router switch connected to at least one CPU) and indirect networks are used. Basic structure of these multiprocessors is depicted in Figure 1.2.

There are two alternative architectural approaches that differ in the method used for communicating data among processors in a distributed-memory system: single address space and multiple address spaces.

In the first method, physically separate memories can be addressed as one logically shared address space, meaning that a memory reference can be made by any processor to any memory location, assuming it has the correct access rights. These multiprocessors are called *distributed shared memory* (DSM) architectures. The term *shared memory* refers to the fact that the *address space* is shared; it does not mean that there is a single, centralized memory. In contrast to the symmet-

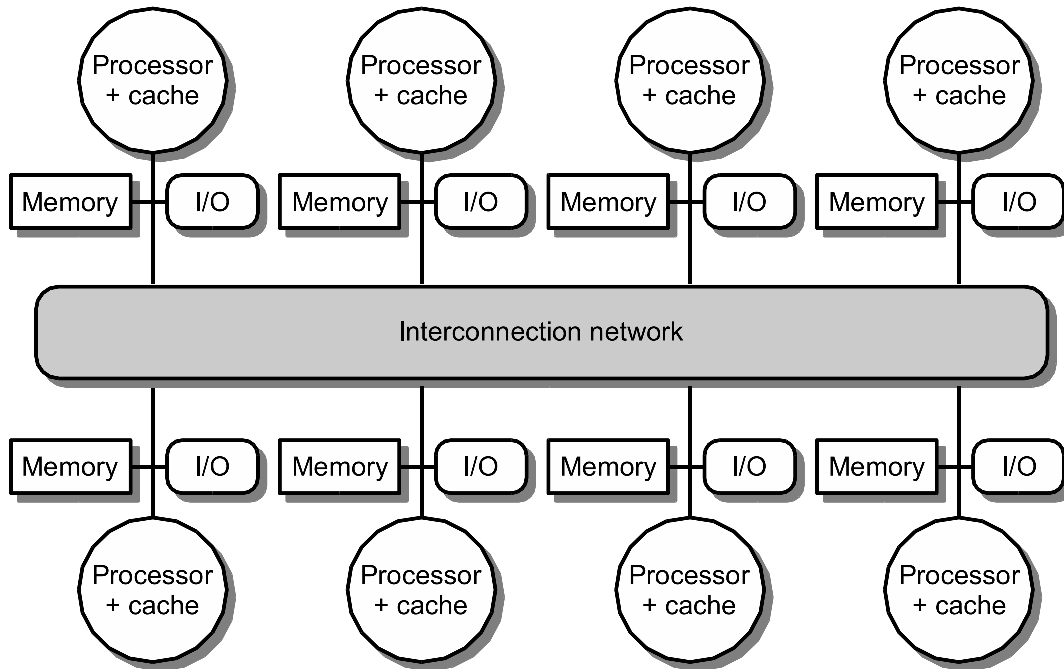


Fig. 1.2: Basic structure of a distributed-memory multiprocessor

ric shared-memory multiprocessors, also known as UMAs (uniform memory access), the DSM multiprocessors use NUMA (nonuniform memory access), since the access time depends on the location of a data word in memory.

Alternatively, the address space can consist of multiple private address spaces that are logically disjoint and cannot be accessed by a remote processor. In such multiprocessors, the same physical address on two different processors refers to two different locations in two different memories. Each processor-memory module is essentially a separate computer; therefore, these parallel processors have been called multicomputers. A multicomputer can even consist of completely separate computers connected on a local area network. It is now widely recognized that a *cluster of workstations* (COW) or *network of workstations* (NOW) offers a very attractive alternative to expensive supercomputers and parallel computer systems for high-performance computing. Using a cluster of workstations has a number of significant and well-enumerated advantages over specially designed multiprocessor systems [6]. Key advantages are as follows:

1. Very high performance workstations and PCs are readily available at low cost.
2. The latest processors can easily be incorporated into the system as they become available.
3. Existing software can be used or modified.

Networks of workstations use standardized or customized interconnect, depending on performance and cost requirements. Two standards are in widespread use, 10 Mbits/sec and 100 Mbits/sec Ethernet (IEEE 802.3), which use coax wire or twisted pair wire interconnects. In addition, a Gigabit Ethernet (1000 Mbits/sec) has

been developed (IEEE 802z). Although a traditional bus topology used in past has been overcome by a star topology with high performance and cost-effective switches, message latency of Ethernet is still very significant, especially with the additional overhead caused by some message passing software. Because the Gigabit Ethernet latency using TCP/IP is about 100 μs [7], such systems are especially suitable for coarse-grained parallel algorithms. Fine-grained algorithms require interconnects with lower latency, such as Myrinet [8].

1.2 Parallel Programming Models

Applications running on parallel computers are written in a certain programming model. In the simplest case, the model consists of multiprogramming a large number of independent sequential programs; no communication or cooperation takes place at the programming level. The more interesting cases include true parallel programming models, such as shared address space, message passing, and data parallel programming. We can describe these models intuitively as follows:

- *Shared variable* programming uses shared data structures, which can be accessed by any processes cooperating on a given task. Individual activities can be orchestrated by accessing the shared data structures.

- *Message passing* is akin to telephone calls or letters, which convey information from a specific sender to a specific receiver. There is a well-defined event when the information is sent or received, and these events are the basis for orchestrating activities performed by individual processes. However, no shared location is accessible to all processes.

- In *data parallel* programming, several processes perform an action on separate elements of a data set simultaneously and then exchange information globally before continuing. The global reorganization of data may be accomplished through accesses to shared addresses or messages since the programming model only defines the overall effect of the parallel steps.

There are several alternatives for implementing these models:

1. Designing a special parallel programming language. An example is a recent project *Orca* [9] or a classical CSP-based programming language Occam [10].

2. Modifying an existing sequential high-level language. Common representatives of language extensions are standards *High Performance Fortran* (HPF) [11] and *Open Multi Processing* (OpenMP) [12], although OpenMP is a hybrid approach, which belongs also to the next category.

3. Using library routines with an existing sequential language. Libraries for message passing are defined for example by standards *Parallel Virtual Machine* (PVM) [13] and *Message Passing Interface* (MPI) [14]. There are several thread libraries used for shared memory programing, e.g. POSIX standard

1003.1c, Java threads and native libraries of some operating systems, but programs usually use these libraries to obtain concurrency for different purposes than achieving speedup in parallel processing.

1.3 Performance Factors

1.3.1 Granularity

To achieve an improvement in speed through the use of parallelism, it is necessary to divide the computation into tasks or *processes* that can be executed simultaneously. The size of a process can be described by its *granularity.* In coarse granularity, each process contains a large number of sequential instructions and takes a substantial time to execute. In fine granularity, a process might consist of a few instructions, or perhaps even one instruction.

Sometimes granularity is defined as the size of the computation between communication or synchronization points. Generally, we want to increase the granularity to reduce the costs of process creation and interprocess communication, but of course this will likely reduce the number of concurrent processes and the amount of parallelism. A suitable compromise has to be made.

For message passing, it is particularly desirable to reduce the communication overhead because of the significant time taken by intercomputer communication. This is especially true for networks of workstations, where the communication latency can be really high. As we divide the problem into more and more parallel parts, at some point the communication time will dominate the overall execution time due to very frequent message passing. The ratio

$$\text{Computation/communication ratio} = \frac{\text{Computation time}}{\text{Communication time}} = \frac{t_{\text{comp}}}{t_{\text{comm}}} \tag{1.1}$$

can be used as a granularity metric. It is very important to maximize the computation/communication ratio while maintaining sufficient parallelism.

1.3.2 Speedup and Its Limitations

Two basic parallel performance measures, which are also used in this thesis, are *speedup factor* and *efficiency.* Speedup $S(P)$ is a measure of relative performance between a multiprocessor system with P processors and a single processor system. It is defined as:

$$S(P) = \frac{t_s}{t_p} \tag{1.2}$$

where t_s is the execution time on a single processor and t_p is the execution time on a multiprocessor.

The (system) *efficiency* E gives the fraction of the time that the processors are being used on the computation. It is defined as

$$E = \frac{t_s}{t_p \times P} \tag{1.3}$$

which leads to

$$E = \frac{S(P)}{P} \times 100\,\% \tag{1.4}$$

when E is given as a percentage.

There are several factors that appear as *overhead* in parallel programs and limit the speedup and efficiency, notably

1. Periods when not all the processors perform useful work and are simply idle (e.g. inherently serial parts of the computation).
2. Extra computation in the parallel version not appearing in the sequential version.
3. Communication time for sending messages in message passing programs.
4. Creation, grouping and management of processes
5. Synchronization of processes.

The simplest performance models take into consideration only the overhead caused by idling (waiting) processors. The fixed workload model, also known as *Amdahl's law* [15], assumes constant serial execution time and is given by the following equation:

$$S(P) = \frac{t_s}{ft_s + (1-f)t_s/P} = \frac{P}{1+(P-1)f} \tag{1.5}$$

where f is a fraction of the computation that cannot be divided into concurrent tasks. Such assumption severely limits the potential speedup — its maximum value according to the equation 1.5 is:

$$\lim_{P\to\infty} S(P) = \frac{1}{f} \tag{1.6}$$

On the other hand, a fixed execution time model assumes that a problem size increases with the number of processors. It defines speedup as:

$$S_s(P) = \frac{s+Pp}{s+p} = s + Pp = P + (1-P)s \tag{1.7}$$

where s is the time for executing the serial part of the computation and p the time for executing the parallel part of the computation on a single processor. s and p also fulfill a condition $s + p = 1$ The equation 1.7 is called *Gustafson's law* [16].

1.4 State of the Art

The need of multiprocessor performance prediction arises not only in computational science in connection with many challenging problems of contemporary science, but also in decision making around multiprocessor software/hardware architecture for embedded systems, recently on a chip [17, 18]. Sound performance evaluation

methodology is essential for credible computer architecture research to evaluate hardware/software architectural ideas or tradeoffs. Performance modeling has to take characteristics of the machine (including an operating system, if any) together with the application software and predict the execution time.

We can classify the different approaches to performance modeling in three categories: analytic modeling, simulation modeling and measurement.

Analytic models are fast and efficient since the behavior is described through mathematical equations. Unfortunately, even if it were possible to have an accurate analytic model of a computer, it would be too complicated. Memory hierarchies, asynchronous events and embedded processor (instruction level) parallelism make it difficult to conceive the model. On high performance computers, which exploit process parallelism, this problem gets even worse, as the performance of the network and the actual communication patterns have often a non-negligible unpredictable effect [19, 20]. In fact, the two often used parallel performance analytic models, *BSP* [21] and *LogP* [22] simplify the parallel architecture to four or five parameters.

A general approach for analytic performance prediction of iterative algorithms running on shared memory systems has been proposed in [23]. Although the model is computationally simpler than many queuing or simulation models [24], unlike them it represents a parallel computation as a sequence of identical loop iterations and does not account for randomness in loop iteration times. It also makes certain assumptions about the modeled architecture, e.g. circuit-switching networks are assumed. Another example of analytic performance prediction technique, which is based on the approximation of parallel flow graphs by sequential flow graphs can be found in [25].

The above approaches require users to explicitly model the application along with the entire system. A source code based analytic performance prediction model for Data-parallel C has been developed by Clement et al. [26]. The approach uses a set of assumptions and specific characteristics of the language to develop a speedup equation for applications in terms of system costs.

Performance prediction of a Fortran program based on source code interpretation has been proposed in [27]. The approach achieves high accuracy of prediction, but it requires the whole program to be implemented, which may demand too much effort in cases when approximate results are sufficient. Moreover, it is restricted to a single programming model and language.

Simulation modeling constructs a reproduction not only of the time behavior of the modeled system but also its structure and organization. It plays an important role in architecture design. Simulation modeling can be more accurate than analytic modeling but is more expensive and time consuming and can be unaffordable for large systems. Therefore, less-detailed simulations that still provide reasonable relative accuracy are required.

As noted in [28], current research of computer architectures tends to simulation-based quantitative evaluation. The authors state that papers on simulation now constitute 80 to 90 percent of the total presented at the International Symposium on Computer Architecture. By comparison, papers based on direct measurements of real systems or on mathematical models have fallen to less than 10 percent from almost 35 percent during the last two decades. Despite this fact, the other methods

still play an important role in computer architecture design. Analytic models, for instance, can help to understand a system in ways that simulation does not provide. They can also be used to validate a simulation-based model.

Generally, it is much more difficult to simulate performance of an application in shared address space than in message passing, since some events of interest are not explicit in the shared variable program (e.g. cache misses requiring bus or network transactions) [29]. In the shared address space, performance modeling is complicated by the very same properties that make developing a program easier: naming, replication and coherence are all implicit, i.e. transparent to the programmer, so it is difficult to determine how much communication occurs and when, e.g. when cache mapping conflicts are involved.

There exist various tools for simulation and prototyping of parallel computations, especially on message passing (MP) systems. Commonly used shared-memory (SM) simulators *rsim, Proteus, Tango, limes* or *MulSim* [30] are not adaptable for MP, although having a single approach, which combines the ability of modeling shared-memory as well as message-passing systems would be undoubtedly valuable.

SystemC modeling environment developed recently [31] is an open initiative directed to specification, simulation and optimization of complex hardware/software systems at higher levels of abstraction. It is a C++ class-based approach that adds multiple concurrent processes, communication and timing to C++ language. SystemC is more like VHDL, suitable for cycle-accurate simulation including some details unnecessary for performance prediction.

System-level modeling and simulation using *time-annotated SDL* specification was attempted in [32] with extra annotations for architecture modeling and the assessment of performance. It enables an exploration of a large number of multi-processor architecture solutions from the very start of the design process, but it considers only one performance factor and lacks accuracy when execution on modern microprocessors with advanced features (pipelining, internal parallelism,...) is simulated.

The Unified Modeling Language (UML) is emerging as a de-facto standard that is widely used for modeling object oriented sequential applications. However, it has also been recently applied to modeling of parallel architectures and programs. An object oriented UML description of the BSP model of parallel computation has been developed [36]. The model can be used for various disciplines of parallel computing research such as parallel architecture, parallel algorithm development, and object oriented implementations of the BSP model on parallel computing machinery.

Extensions to UML which support modeling of performance-oriented parallel and distributed applications were presented in [37]. The project includes a set of UML building blocks that model the most important constructs of message passing and shared memory parallel paradigms. The UML models support further annotation with performance information, which can be used for performance prediction. However, a performance estimation tool based on the UML models [38] is still under development.

Performance analysis is another important activity that requires parallel program code already running on a target platform. Empirical experimentation is an irreplaceable task in any science. Measurement methods allow to identify bottlenecks

of a real parallel system performance. This approach is often expensive because it requires special purpose hardware and software. Performance measurement can be highly accurate when a correct instrumentation design is carried out for a target machine. It is a cyclic process of measuring and analyzing performance data, which is driven by the programmer's hypotheses on potential performance problems. But, as in any science field, real systems are not always available.

Various tools for automatic performance analysis and tuning have been developed. A recently suggested approach based on performance property specification language [33] formalizes performance bottlenecks and the data required to detect them. It has been used to formalize properties of parallel programs using various programming paradigms such as OpenMP [34] and MPI [35]. The specification language is similar to Java but uses only single inheritance and does not require methods. It uses a special syntax to specify performance properties. The models are highly dependent on a target architecture's support for performance data measurement (e.g. counters of cache misses, remote page accesses, etc.). The approach is general enough to support various programming models and architectures. An automatic system that provides automatic performance analysis based on the specification language is currently being developed.

1.5 Dissertation Goals and Used Methods

A primary goal of this thesis is development of a unified approach to modeling of parallel architectures and algorithms with special emphasis on estimation of obtainable performance. The modeling language and simulator *Transim* [39] has been chosen as a base tool. Although it was designed by its authors just as a transputer simulator for prototyping and performance evaluation of message-passing programs, the goal was to apply it to simulations of many different types of parallel architectures and programming paradigms. This goes far beyond the originally anticipated Transim applications.

The selected approach should be able to provide means for simulating theoretical parallel machine models like PRAM or APRAM as well as real shared-memory and distributed-memory machines with various underlying communication architectures (memory hierarchy, direct and indirect interconnection networks). It should also support simulation of different types of parallel programming models (message-passing, shared-variable, data-parallel). Development of a uniform description of parallel software and hardware on distributed- as well as shared-memory architectures would allow fair performance comparison, especially when message passing and shared variable programs may run on either architecture. Moreover, the approach should achieve acceptable relative accuracy of simulation, which can be proved by comparison of simulation results with real executions on available architectures.

1.5.1 Transim Description

Transim is based on a formal approach to concurrency known as Communicating Sequential Processes (CSP), originally devised by Prof. C. A. R. Hoare [40], which can be used for modeling computer hardware. Hardware modules are represented

as sequential processes executing in parallel, and their interaction is described by passing messages among them. Unidirectional point-to-point static communication channels are defined in CSP together with basic channel operations *receive, send, select* (`?`, `!`, `ALT`). The CSP is also used in teaching concurrency [41, 42]. A well-known implementation of CSP is *Occam* language [10], which was developed in concert with a transputer processor. The Transim language is a subset of Occam with certain extensions required for performance evaluation.

The Transim simulator runs under Unix or MS DOS. Simulation is based on accumulation of time intervals t_{comp_i} of useful computations (which are not in fact carried out) and separately time intervals t_{over_i} of nonproductive overhead activities (idling, communication, waiting for a partner or for a timer). Since parallel time T_P is the same for all P processors and sequential time T_S is taken as a sum of all periods of useful computation in all processors, the simulator derives speedup S and average efficiency E as

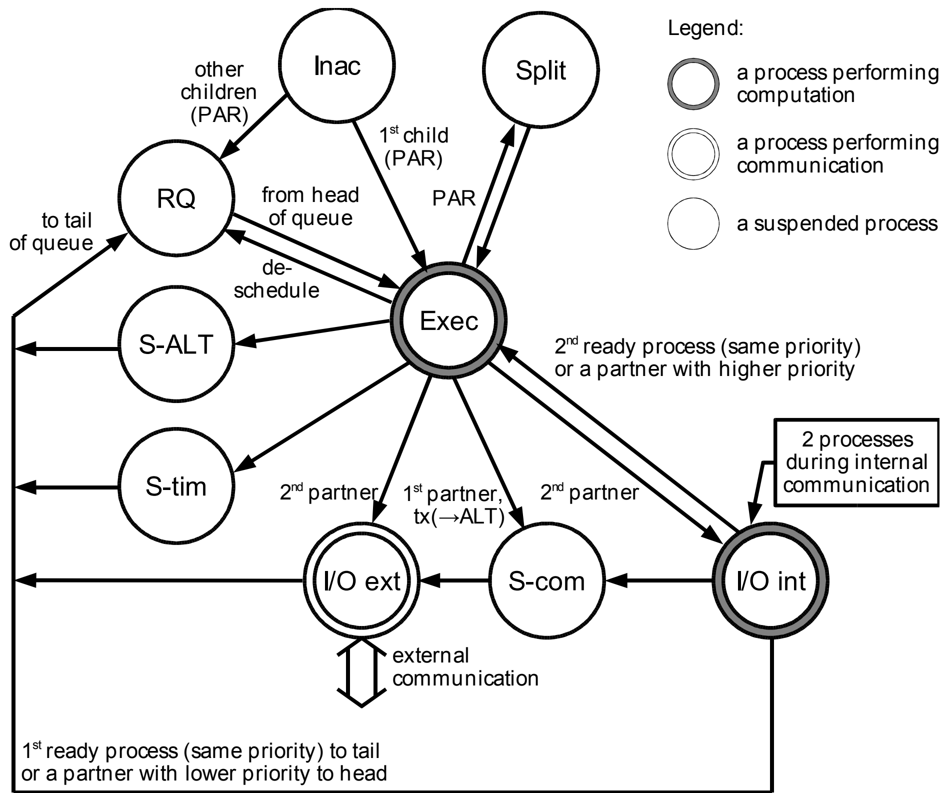
$$S(n,P) = \frac{T_S}{T_P} = \frac{1}{T_P}\sum_{i=1}^{P} t_{\mathrm{comp}_i} = \sum_{i=1}^{P} \frac{t_{\mathrm{comp}_i}}{(t_{\mathrm{comp}_i} + t_{\mathrm{over}_i})} = \sum_{i=1}^{P} E_i \tag{1.8}$$

$$E = \frac{S}{P} \tag{1.9}$$

The Transim language used to write input files for the simulator is not a full-fledged programming language, but only a descriptive language suitable for specifications of parallel software, hardware and mapping to one another. It contains only predefined procedures and functions and limited data types. Software description contains all point-to-point channel communications and control statements, whereas the sequential code is replaced by timing constructs `SERV()`. An argument of `SERV()` specifies the number of CPU cycles taken by the task. Data-dependent computations can also use a random `INT` value in a certain interval `RAND(low, high)`. Integer computations can be carried out, but are timeless. Often they can be used for small demonstration examples running simultaneously with simulated parallel execution of a much larger problem. Real data types are not supported at all. Explicit overhead can be represented directly by `WAIT()` construct.

Process creation is done either by replicating the same code in a subset of processors using replicated `PLACED PAR` construct (SPMD model), or using different codes for processes running on different processors (MPMD model). Several parallel processes on one CPU are created by `PAR` construct. Using macros and C preprocessor can avoid copying the same code many times (e.g. barrier code). Some features of a distributed kernel are also built in Transim, namely process priorities and scheduling. Processes in Transim are static, each process code is denoted by `SEQ | process.name`. Two priorities, high and low, are defined for processes running on the same processor. Whereas the low priority processes are timesliced, high priority processes run until blocking condition occurs (communication, waiting for a timer, etc.). Timers can be declared with two resolutions according to the priority level. Transim process state and transition diagram is shown in Figure 1.3.

As far as communication is concerned, unidirectional point-to-point channels (declared as `CHAN OF ANY channel.name`) are used. Communication overhead based on a linear communication model (set-up time and transfer time proportional to the



Legend

Inac	Inactive processes
Split	A parent process that executed the `PAR` statement
RQ	Ready Queue — processes ready for execution
Exec	Currently executing process
S-ALT	A process waiting for a partner in the `ALT` statement
I/O ext	A process communicating with a process running on a different processor
S-com	A process waiting for communication
I/O int	A process communicating with a process running on the same processor
S-tim	A process suspended for a time interval determined by a timer

Fig. 1.3: State diagram of process transitions in Transim

message length) can be defined by the user. The simulated messages can be of arbitrary length (up to `MAXINT` bytes). The length of the message in bytes is specified in the output statement, but only a single `INT` is actually transferred in a channel. This is not as restrictive as it seems; longer messages may be put in a shared memory and retrieved in no time by the receiver process. Interprocess communication in Transim is unbuffered and uses implicit synchronization: both the partners (processes) wishing to communicate must be ready, i.e. synchronization is achieved at each communication. Other types of communications (non-blocking, asynchronous, collective, WH routing, etc.) can be built up using available primitives receive, send, select (`?`, `!`, `ALT`).

The architecture and CPU parameters are specified by `NODE` construct in hardware description (clock speed, link speed, communication latencies etc.), otherwise the default values are used. Any piece of hardware, not only CPU, can be simulated. Channel placement on physical links, two bi-directional channels per link, is done automatically, the simulator extracts topology from the Transim file. The number of physical links per processor is not limited. Only if different speeds are required on different channels, then topology must be described by `LINK` statements. The mapping between software and hardware is made through the `MAP` construct, several processes can be mapped on one CPU and will run in parallel.

Transim simulator generates quite a few output files, various statistics on utilization of processors, time spent by processes and channels in all possible states. Average processor utilization and speedup are given first, equations 1.8 and 1.9. Processors can send messages/data to a report during simulation, either by using `NOTE(note.text)` which prints a message, a process id and simulated time, or by predefined channel `OUTSTRM` (one per processor) e.g. `OUTSTRM ! data1, data2,...` which will print intermediate results.

A further reference to Transim language constructs, which may be useful for better understanding of code examples in this thesis, can be found in appendix A.

1.6 Overview of the Following Chapters

The first part of this chapter presented an introduction to parallel processing and its overhead, which determines a resulting performance. A short description of widely used parallel architectures and programming models was given in sections 1.1 and 1.2. Next, the overview of a state of art in the specified research area was given and, finally, goals of this thesis were introduced together with methods how to fulfill them.

Simulation of theoretical models of parallel systems is described in chapter 2. Features and limitations of the most frequently used abstract models are given first. Then, simulation of a finite size problem execution in Transim is described. The approach taken is demonstrated on an example of tuning performance of a parallel sorting algorithm.

Chapter 3 describes cache-coherent bus-based symmetric multiprocessors. Descriptions of models of most frequently used synchronization primitives like various locks and barriers are also given. Models of these primitives can be used as building blocks of more complex shared-variable algorithms. Achieved accuracy of simulation is demonstrated by comparison of simulation and real execution of a parallel Fast Fourier Transform algorithm.

Modeling of cluster computations is presented in chapter 4. At first, descriptions of various cluster interconnections like Omega network, fat tree, crossbar switches and different switching techniques are given. Then, a model of a SMP cluster is introduced in section 4.2. It combines the cluster interconnection models with a SMP model described in chapter 3 to form a more complex model of an architecture suitable for a hybrid message-passing/shared-variable programming paradigm. Results of simulations are again compared to real executions on a cluster of workstations.

Gauss-Jordan elimination method for solving large systems of linear equations is used as a benchmark algorithm.

Performance modeling and design of iterative and optimization algorithms is described in chapter 5. The developed iterative algorithm GAroute is intended for optimization of group communication on irregular topologies. Design of both sequential and parallel versions is presented together with simulation-supported performance tuning of the parallel implementation. Presented results indicate ability of the designed algorithm to provide results useful also in other areas of parallel computing.

Finally, chapter 6 summarizes main contributions of the thesis and proposes possible directions of future research.

Appendix A contains a short reference of the most frequently used Transim commands. It can be useful for better understanding to code examples presented in the thesis.

Chapter 2

Abstract Models of Parallel Systems

Significant variety of parallel computer architectures caused that several mathematical models of parallel computer systems have been developed to aid in easier design of algorithms and theoretical analysis of behavior, complexity and correctness of parallel computations. Suggested models include e.g. PRAM, APRAM or BSP abstract machines [44].

Performance analysis of algorithms designed for such models is oriented mainly to asymptotic complexity in time or space and it may not often reflect properties of finite size of problems on real machines with limited resources and many sources of overhead, which are difficult to include in the theoretical models. Theoretical study of performance oriented only to asymptotic complexity has not much in common with real machine performance. There are two reasons for that: an arbitrarily large hidden multiplicative constant in asymptotic time complexity expressions and a requirement that problem size $n > n_0$, $n \to \infty$. An asymptotically superior algorithm may often be worse than the one asymptotically inferior for the whole range of n of practical interest.

Features and limitations of the theoretical models are discussed in the following sections. Then, it is shown how execution of algorithms designed for the abstract models can be simulated in Transim to find out their performance for a given problem size. Performance tuning of a parallel sorting algorithm is also undertaken, using two modifications of the basic algorithm. The real-time performance of PRAM programs is evaluated directly by Transim simulator based on timing constructs in software description.

2.1 PRAM

Parallel Random Access Machine (PRAM [43]) is a simple model of a single instruction/multiple data symmetric multiprocessor computer (SIMD SMP). It is an extension of a classical model of a sequential computer *Random Access Machine* (RAM [44]).

The RAM model consists of a computing unit, input tape and output tape. There is no limitation on number and/or sizes of memory cells. *Time complexity*

is defined as a number of executed instructions since execution of all instructions takes a unit time regardless to lengths of its operands. *Space complexity* is defined as a number of accessed memory cells.

A parallel model (PRAM) consists of an arbitrary number of processors connected to a common memory. All processors are implicitly synchronized. Every processor can access any cell of the shared memory in a unit time. Processors communicate only by means of the shared memory. *Parallel time complexity* is equal to the time of execution of a program. *Space complexity* equals to a number of accessed shared memory cells.

PRAM contains many simplifications of parallel computers, which make it attractive to parallel algorithm designers, but which, at the same time, make the model too unrealistic for reliable prediction of performance of the parallel algorithms on real machines. The limitations and details of lower levels of architectures ignored by the model includes e.g. conflicts and overhead of accesses to a shared memory, synchronization overhead, connectivity, speed limits and bandwidth capacity of interconnection network's links etc. To bring the model as near to real architectures as possible, finite number of memory cells and finite word length of PRAM were assumed in simulations. Moreover, the most restricted model of concurrent access to shared memory, EREW (Exclusive Read, Exclusive Write), was used.

2.2 Asynchronous PRAM (APRAM)

Real computers differ from PRAM models in several properties. Most of MIMD computers and their processors do not run synchronously. Beside that, access to shared memory takes longer than local operations on registers. Therefore, parallel models which better reflect properties of real computers with shared memory were designed. An elegant model is APRAM, which differs from PRAM in several points:

- Processors work *asynchronously*; there is no central clock.
- Processors must *explicitly* synchronize.
- Memory access time is *not unit*.

Explicit synchronization is performed by *barrier* operations, which are logical points in code of a given set of processes. Every process stops at this point until other processes reach the point. Then all of them continue.

The computation on an APRAM computer is defined as a sequence of global phases, in which processors work asynchronously, divided by barrier synchronization. A state of computation (contents of shared memory) can only be determined at the place of a barrier. The state cannot be determined at any place between barriers because it depends on relative speeds of processes, which are neither defined nor determinate. Therefore, every APRAM algorithm is constrained with an important condition:

Rule 1
If a memory cell is written by a certain processor in a given global phase, it must not be accessed by any other processor in the same phase.

2.3 Bulk Synchronous Parallel (BSP) Model

BSP is a more realistic model of a MIMD computer with distributed memory [21]. It is very similar to APRAM, except that collective communication operations are used instead of reads and writes from/to a shared memory. A computation consists of so called *supersteps*. A superstep is defined as a set of independent local computations followed by a global communication phase and a barrier synchronization step. The model consists of:

- A set of processor-memory pairs.
- A communication network that delivers messages in a point-to-point manner.
- A mechanism for the efficient barrier synchronization for all or a subset of the processes.
- There are no special combining, replicating, or broadcasting facilities.

Some of BSP's fundamental properties are that programs are simple to write, the model is independent of target architectures, and the performance of a program on a given architecture is predictable. This is achieved by considering computation and communication at the level of the entire program and the executing computer.

2.4 Simulating PRAM Algorithms with Transim

The main shortcoming of the PRAM model lies in its unrealistic assumption of zero memory latency and instruction-level synchrony. Nevertheless, there were several attempts to develop PRAM simulator, especially at universities, enabling practical PRAM programming. One of such examples is a simulator and language Fork as reported in [45].

An alternative approach described in this section and published in [63] is based on the Transim simulator. This approach has been chosen because it supports comparison of finite problem size PRAM simulations to simulations of real machine executions with reasonable processor count P. In addition, it leaves the speed vs. accuracy tradeoff on the user, who can control the level of detail and accuracy of simulation.

In order to make the PRAM model resemble more closely to modern RISC processors in real machines, the most constrained variant EREW APRAM has been chosen for simulation and load and store instructions are separated from compute instructions, which operate on register operands only. Each register instruction executes in one clock cycle, and all processors synchronize at instruction level. This corresponds to IPC $= 1$, which is used in simulation of real architectures, too.

The simulated algorithm was the bitonic sort, for more than 20 years the best sorting algorithm for hypercubes, with time complexity $O(\log^2 n)$ when sorting n keys on $P = n$ processors. The algorithm is implemented as a single program/multiple data (SPMD) application. In general, shared memory access can take more than 1 clock cycle (actually the first access in a block of data d clocks, other

accesses 1 clock, since a pipelined load/store unit is assumed) and a MPMD model of processing can also be used, in which case synchronizing barriers are needed. All these generalizations lead to the APRAM model and are easily described in Transim.

Although Transim is naturally intended for message-passing distributed memory systems, it supports also shared variables. This property enables successful simulation of a PRAM model as well. Communication operations are not needed for this model at all.

A multi-port shared memory read or write by client processes is simulated by read or write by one process at a time. Unless two or more processors write into the same memory location, we do not have ambiguity. We must therefore make sure that this situation does not occur, because simulation would not stop in this case, leaving in a memory location the value written last.

PRAM machine synchrony at instruction level is obtained using `SERV()` or `WAIT()` statements with correct arguments inside a (conditional) code. In case of APRAM, a simple model of a sense-reversal barrier (Figure 3.6) as described in section 3.2.2, was used for synchronization. The model is suitable for repeated use in loops and the synchronization overhead can be set explicitly.

2.4.1 Performance Tuning of a Sorting Algorithm

Sorting of n keys using compare-and-exchange operations has sequential complexity $O(n \log n)$ and belongs to NC-class of problems solvable in polylog time on PRAM [46]. In what follows we will consider the bitonic sort only, which can sort n keys in $O(\log^2 n)$ steps.

If the number of keys n is larger (sometimes by several orders of magnitude) than processor count P, then each processor does local sort of $\frac{n}{P}$ keys followed by the repeated bitonic merge and split of 2 sorted sequences. The length of the working sequence handled by each processor in every step is thus constant and equal to $\frac{n}{P}$. Bitonic merge and split is done $(1+2+3+\ldots+\log P)$ times. In each merge operation two sorted sequences of $\frac{n}{P}$ items (blocks of consecutive data in a temporary array of n elements in a shared memory) are merged into one sorted sequence, which is then split into upper and lower parts. The two parts are rearranged within the temporary array. Complexity of this bitonic sort of n keys on P processors is thus:

$$O(n, P) = \frac{n}{P}\left(\log\frac{n}{P} + \log^2 P + \log P\right) \quad (2.1)$$

But the hidden multiplicative constant in big O notation can be large or small depending on a particular implementation.

The decision which two sequences are merged by each processor in each step is made according to rules of the bitonic sorting algorithm. However, implementations can differ. The straightforward naive approach (denoted *sort* in further paragraphs) is e.g.:

1. Node i and node j exchange copies of sorted subsequences.
2. Each node merges its subsequence with the received one.
3. Each node keeps its half and throws away the other half.

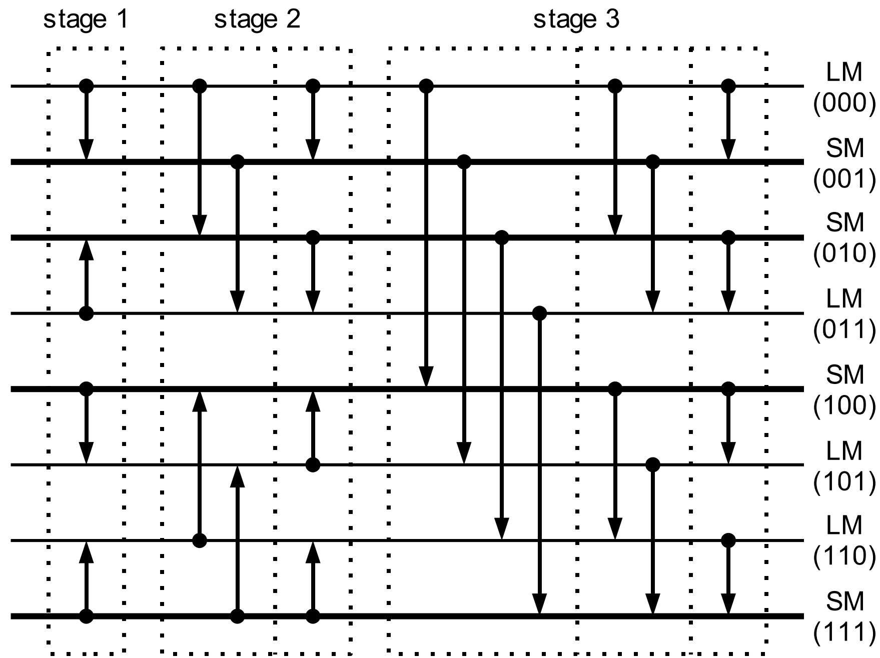


Fig. 2.1: Bitonic sort algorithm

If the merging is done by one node only and then a relevant half of the resulting merged sequence made available to another node (that was idling during the merge operation), no time will be saved.

A performance of the algorithm can be improved if one of the processors i and j starts merging from the top, whereas the other one from the bottom and each does exactly one half ($\frac{n}{P}$ keys) of the resulting sequence. No processor will throw away a half of its work, and each processor will do a lower or an upper part depending on which part it is going to keep. The length of communicated sequences will remain the same as in the original algorithm, $\frac{n}{P}$ keys in each step. This variant of the bitonic sort will be denoted *sort+*.

Another version of a faster bitonic sort reduces the number of shared-memory communications (when one process writes a shared data object and another process reads it) [47]. This communication can be very time-consuming in case of real shared-memory computers due to frequent cache misses. The reduction of communication is achieved by keeping one half of a merged sequence in a local memory (supposing the local memory is large enough to hold the sequence) and storing only the other half in a shared memory.

In contrast to the straightforward algorithm, $2P$ sequences (each of size $\frac{n}{2P}$ elements) are merged by P processors instead of former P sequences (each of size $\frac{n}{P}$). At first, each processor sequentially sorts two of the sequences. Because the sorted sequences are now shorter, bitonic merge and split is done $(1+2+3+\ldots+\log 2P)$ times, which is more times than before. Since one of the two sequences produced by the merge and split operation in each step can be used by the same processor for merge and split also in the next step, it can be kept in a local memory.

Figure 2.1 illustrates how the merge and split operations are arranged. The example shows sorting a sequence of n elements by $P = 4$ processors. The input

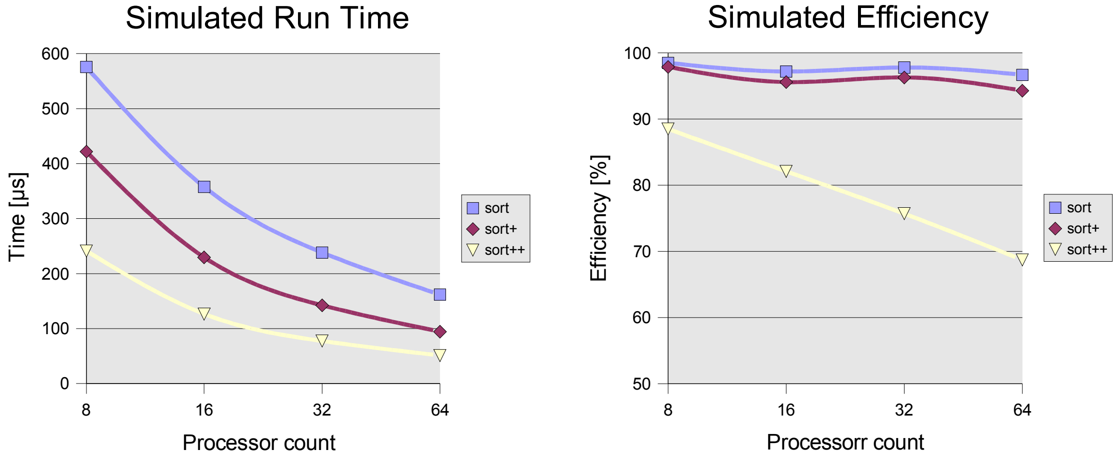


Fig. 2.2: Simulated run time and efficiency of 3 variants of bitonic sort

sequence is divided into eight short sequences of size $\frac{n}{8}$, which are represented by horizontal lines. Thick lines denoted *SM* represent sequences stored in a shared memory and thin lines denoted *LM* represent sequences stored in local memories of processors. The distinction is performed according to a parity strategy: a sequence whose binary index (shown in parentheses in the figure) has an even number of 1's is stored in a local memory, others are stored in a shared memory.

Dotted lines in Figure 2.1 depict three different stages of the algorithm and steps performed in each stage. The four pairs of sequences which are merged in each step are designated by vertical lines. Arrows point to locations where higher halves of the merged sequences are stored.

2.4.2 Results of a Simulation

Figure 2.2 shows time complexity and efficiency of the three mentioned algorithms simulated in Transim. Sorting of 1024 32-bit integers was simulated for a processor count varying from 8 to 64. The only source of inefficiency in APRAM processing is load imbalance — the fact that not all processors are busy all the time. The cost of message passing or shared memory communication and of other overheads in real machines is not taken into account.

It can be seen that Transim approach aids prototyping parallel algorithms as well as their performance evaluation, so that comparison of time complexities of various versions of an algorithm can be easily performed. In each simulation run, the small size demonstration example is running simultaneously with a full-scale simulation. This is made possible by the fact that simulated time is advanced only by user inserted constructs `SERV()` and `WAIT()`. The execution of a demonstration program does not consume any simulated time at all.

To demonstrate how assumptions of abstract models can differ from properties of real machines, another simulation was performed. Two simulation models were compared: PRAM model and a bus-based shared-memory multiprocessor (bb-SMP). The bb-SMP model will be described in more detail in chapter 3, where attainable accuracy of simulation will also be demonstrated by comparison of a simulated and real execution.

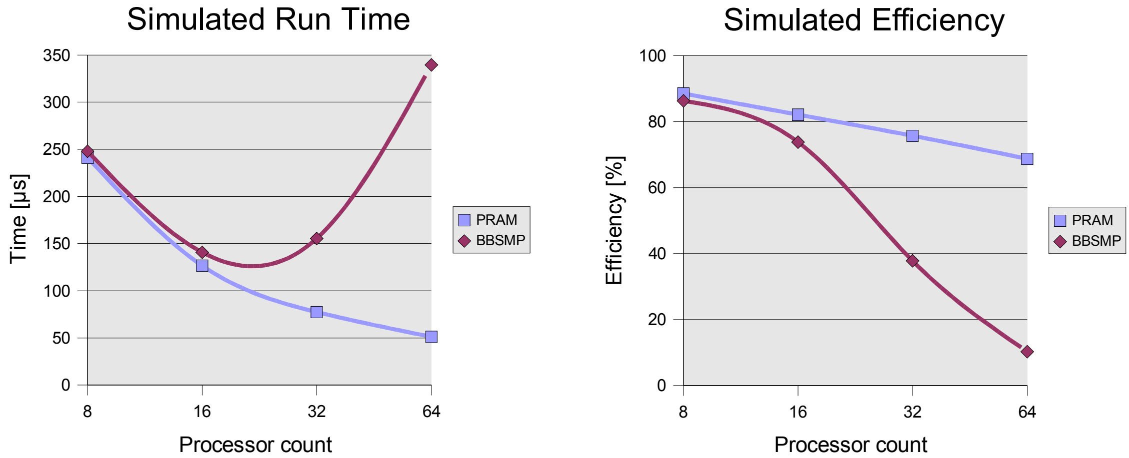


Fig. 2.3: Comparison of simulated run time and efficiency of sort++ on PRAM and a bus-based SMP

Parameters of the bb-SMP were similar to PRAM, but instead of the assumption that access to any shared memory cell takes a unit time, cache misses together with corresponding communications (bus transfers) were simulated. Execution of the third variant of bitonic sort (*sort++* as described above) was simulated with both models. Results of the simulations are plotted in Figure 2.3. It is apparent that the bb-SMP machine suffers from considerably worse scalability than the PRAM model. It is due to finite bandwidth of the shared bus, which is already saturated when more than 16 processors are used for the given task.

2.5 Conclusions

This chapter presented an innovative application of APRAM model. Instead of studying asymptotic complexity (problem size $n \to \infty$), which is a common application of abstract machine models, prediction of execution times of real parallel programs was performed with realistic numbers of processing units and parameters of modern microprocessors.

Results of simulations indicate that the used approach can be useful not only to design and tune performance of a given algorithm on an abstract machine, but also to evaluate what performance can be expected when the implemented program is run on a real machine. Since the algorithms can be simulated not only on APRAM, but also on many real architectures like hypercubes, symmetrical multiprocessors, clusters etc., this approach makes comparison of practical computations on abstract and real machines possible.

Chapter 3

Bus-based Symmetrical Multiprocessors

Parallel computer systems are at present dominated by shared memory (SM) multiprocessors and by clusters of workstations or PCs. Recently, the convergence of these systems has led to scalable distributed SM architectures supporting both shared variable and MP programming models [29]. This fact makes it reasonable and useful to develop a single simulation system for both kinds of architectures and programming paradigms.

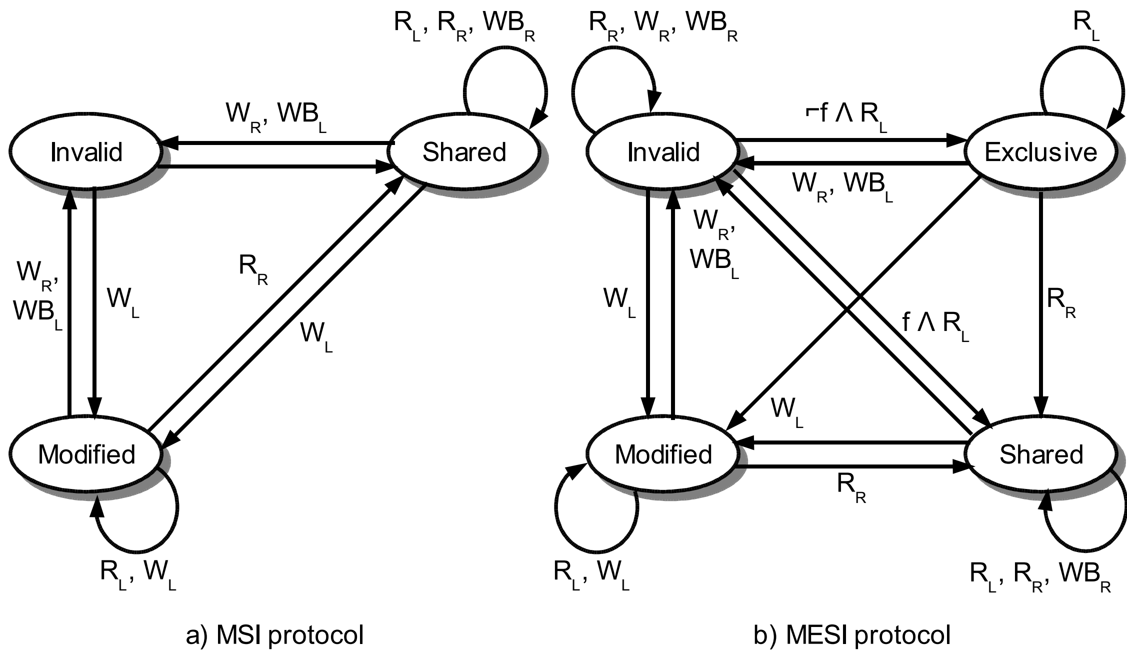
Although there have been several tools developed, which support performance analysis of a variety of programming paradigms and architectures, e.g the APART project [33], they are determined for analysis of performance data obtained from executions on target architectures. Since they require an algorithm to be fully implemented in a given programming language and executed in the chosen environment, such tools are not suitable for performance prediction of algorithms in early design stages. Moreover, it is not possible to make a comparison of a program's execution on parallel architectures, which are not available.

This chapter presents an approach to shared memory architecture modeling which is based on the Transim simulation tool and language. At first, an active monitor-like model of a cache-coherent bus-based (CCBB) multiprocessor is introduced. Then, modeling of synchronization primitives like locks and barriers, which are used as building blocks of many shared variable programs, is described. Finally, accuracy of the model is illustrated by comparison of data from simulated and real execution on a shared memory machine.

3.1 Modeling CCBB Multiprocessors

When a shared memory multiprocessor is simulated, several parameters of the architecture which influence its performance must be considered depending on a level of detail of the simulation. They include e.g. bus width and frequency, type of transactions (connected or split), memory update policy, cache coherence protocol, etc.

A system with a *connected (atomic) transaction* bus allocates the bus for a next transaction only after a previous transaction finished. On the other hand, *split*



Legend

R	Read request
W	Write request
WB	Write back request
index $_L$	Request issued by the local CPU
index $_R$	Request issued by any remote CPU
f	Sharing detection function: returns *true* if a valid copy of the requested block exists in a cache of any remote CPU.

Fig. 3.1: MSI and MESI cache coherence protocols

transactions used in some high performance systems allow multiple bus masters at a time. They achieve a higher throughput at a cost of higher latency due to multiple bus arbitration.

Coherence actions in a cache coherent system can be performed in different ways. A *write invalidate* method, which transfers data in whole blocks, is more popular than *write broadcast*, which works with individual words. The memory update can be performed either immediately after a write operation (*write through* method) or as late as possible to achieve better performance (*write back* method).

A cache coherence protocol determines when bus transfers in a shared memory system occur. State diagrams of two popular protocols MSI and MESI are depicted in Figure 3.1. The diagrams determine how states of a data block in local cache of a particular processor change. Due to an extra state *Exclusive*, the MESI protocol provides better performance than MSI when write hit occurs, since no data transfers/invalidate commands occur if the block to be written is exclusively owned by the cache of the local CPU.

In further paragraphs we will consider updating SM by the write back method and the write invalidate variant of a cache coherence protocol. Other alternatives

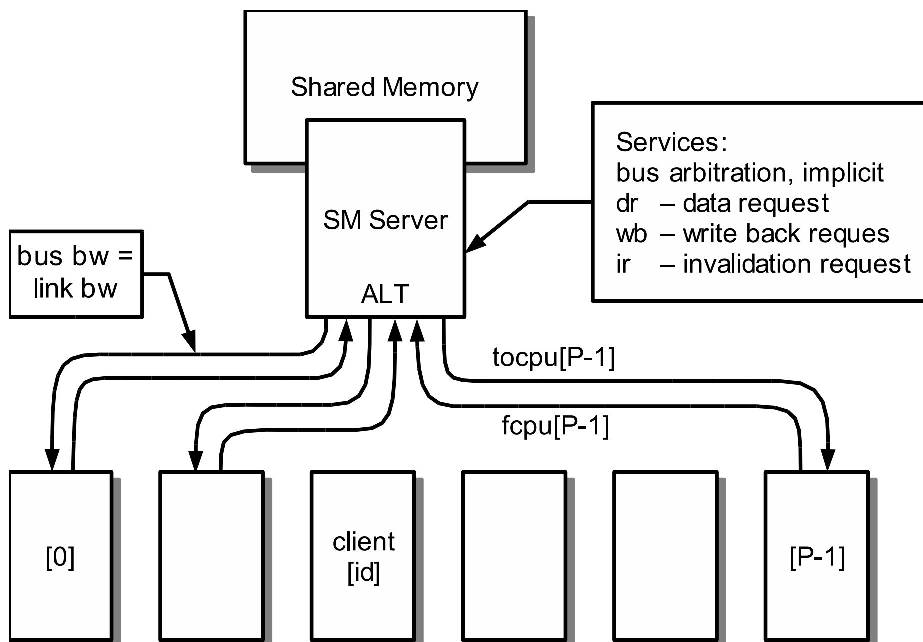


Fig. 3.2: Clients/SM server model of a cache-coherent, bus-based, SM multiprocessor

(write through, write broadcast) can be dealt with similarly. A main multiprocessor memory and the bus (in the case of an atomic bus) are shared resources that have to be exclusively allocated to a client process running on a certain CPU. Further we will describe a model of an atomic bus, which could be extended to support also other possibilities like pipelined or split-transaction buses.

We can find some resemblance in behavior of shared memory and a bus with behavior of *monitors* used for communication and synchronization in concurrent programs written e.g. in *Java* [48]. Monitor procedures (methods) execute in mutual exclusion and the same is true about data requests (`dr`) or write back (`wb`) requests generated by individual processors. We do not need to treat read and write misses in local caches differently in simulation, because both of them are satisfied with data requests; the read as well as write operations are always carried out in a local cache after a valid copy of the block is obtained. If a cache is full, and if a modified block is selected for replacement, it has to be written back to the main memory before it is overwritten.

Using duality between monitors and message-passing [49], we can simulate a monitor by a client and a server, with operations `dr` and `wb` as shown schematically in Figure 3.2. A simplified Transim code of the model is in Figure 3.3. We will refer to our active monitor as to the *SM server*. There are still some missing features in it. A write hit to a shared copy by one CPU must generate invalidate messages on the bus, which should reach relevant processors simultaneously. Bus occupancy by the invalidation message is easily represented by a special invalidation request (`ir`) to SM server, which blocks bus utilization for an appropriate number of cycles. Broadcast of invalidation is ensured by incrementing a shared variable (`block_state`) by SM server. Processors can recognize an obsolete copy of the block if the just visible value of the shared variable `block_state` differs from the local copy of the same variable.

```
[P] BOOL busrq; INT nowserving:  -- shared variables
-- array of channels from cpu[] and to cpu[]
[P] CHAN OF ANY fcpu,tocpu:

PLACED PAR
  INT x,j,q,ptr:
  SEQ | SMserver  -- SM server and bus arbitration process
    WHILE TRUE
      SEQ
        IF
          arbitration = a_fair
            j := (q+1) REM P  -- fair bus arbitration everytime
          arbitration = a_fair_P
            j := (j+1) REM P  -- fair bus arbitration for P requests
          arbitration = a_priority
            j := 0            -- multiple requests, fixed priority
          arbitration = a_random
            j := RAND(0,P-1)  -- multiple requests, random selection
        -- In fact replicated ALT is not supported by Transim.
        -- It should be expanded to individual branches.
        PRI ALT i = 0 FOR P
          fcpu[(j+i) REM P] ? x
            q := i               -- client's ID

        IF
          x = dr              -- data request
            tocpu[q] ! block_state | blocksize  -- data response
          x = wb              -- write back request
            WAIT(smtime)
          x = ir              -- invalidation request
            WAIT(itime)

PLACED PAR d = 0 FOR P        -- P client processes
  INT copy:
  SEQ | client
    PRI PAR
      SEQ | high              -- in high priority atomically
        fcpu[d] ! wb | wbsize -- replace old data to SM if necessary
        fcpu[d] ! dr | drsize -- request new data
        tocpu[d] ? copy       -- receive new data
      SEQ | low
        SKIP
```

Fig. 3.3: Clients and SM server with three operations, software description

Transim as an extended subset of Occam 2 supports synchronous channel communication (a linear time model) and is naturally intended for message-passing distributed memory systems. If it is used for simulation of CCBB systems, then bus transactions modeled as communications between node processes and a central SM server process running on an extra processor have to be of equal duration as message passing communication. This can be easily arranged by selection channels' speed and setup time and/or using additional delays `smtime`, `itime` as in Figure 3.3.

3.1.1 Fairness and Overhead of `ALT` and Arbiters

Another feature of the SM server which deserves more comments is a bus allocation strategy — in other words, a model of a bus arbiter. Implementation of a bus arbiter in software should be flexible enough to allow fair, priority, random or other arbitration strategies. The bus arbiter should process pending requests for the bus according to a selected strategy.

The code of a bus arbiter (`SMserver` process) in Figure 3.3 contains an `ALT` statement. It is used in a replicated form for simplicity although Transim supports only expanded forms, so that all inputs would be enumerated in an actual simulation model. The main function of the `ALT` statement is serialization of bus requests. Unlike an implementation of `ALT` in Occam, which has an implicit overhead depending on a number of inputs, in Transim the statement does not have any implicit overhead. Nevertheless an overhead of a hardware arbiter can be added explicitly by using a `WAIT` statement.

Although inputs of a `PRI ALT` statement used in the model in Figure 3.3 are always checked in textual order for readiness to communication, the actual order of channels can be modified by changing an indexing scheme. This is performed by the addition of variable `j` value. The variable is assigned a new value in every loop iteration depending on the actual arbitration policy. Increasing the value by one as sometimes mentioned in literature may provide fair bus arbitration but only in the case when all P processors request the bus. When only a subset of processors continuously request the bus, some of them might succeed more often than others. This variant is denoted `a_fair_P` in the figure.

The approach was modified to provide a model of a truly fair bus arbiter. It is denoted as `a_fair` variant. Instead of skipping only the first channel, it also skips all following channels to avoid reading from the same channel more than once in a sequence when other channels are ready as well. Although this model already guarantees fairness of arbitration it could be further extended to maintain requests in the order in which they are issued. This requires use of a circular queue of requests. Each requester would write its index to the tail of the queue and a bus arbiter would serve requests from the head of the queue.

3.2 Models of Synchronization Primitives

There are three basic synchronization mechanisms in shared variable programs:

- atomic sequences
- critical sections with mutual exclusion
- condition synchronization.

Atomic sequences are not directly supported in a typical processor hardware. For very short atomic sequences one can use either atomic instructions of the type `read-modify-write` or pairs of atomic instructions `LL-SC` (`Load Locked-Store Conditional`). Longer atomic sequences can be sometimes implemented as high-priority processes or, not the best way, by mutual exclusion. Critical sections are executed atomically, but atomic sequences do not require exclusive access. Critical sections are typically implemented by locks, condition synchronization by shared flags or counters in case of two processes and by barriers in general case.

The models developed for simulation of shared variable programs and published in [64] include the most frequently used types of locks and barriers:

- simple locks based on atomic `read-modify-write` instructions
- array lock
- ticket lock
- centralized sense-reversal barrier
- butterfly barrier
- dissemination barrier.

3.2.1 Locks

Simple lock implementations assume a hardware support (atomic `Test & Set`, `LL-SC` instruction pair), yet they create a huge traffic on the bus because of perpetual effort to get the exclusive copy of the lock for modification, which leads to bus congestion. Testing a copy of a shared lock variable in a local cache using a load instruction only and trying `Test & Set` as soon as the lock copy becomes invalid reduces the bus traffic since invalidation messages are not generated during every access to the shared cache block (so called `Test-and-Test & Set` operation). Despite this improvement, the overhead is still large (proportional to P^2) when several processes want to acquire the lock simultaneously [29].

The model of this lock with fair bus allocation strategy uses a buffer `[P] BOOL busrq` to register bus requests. The buffer is scanned in circular fashion until the first request is found. A requester is notified through a shared variable `nowserving`. Its request is serviced right away and then the next request is found and serviced, etc. When the process loads a fresh copy of the lock, three situations may occur, as shown in a simplified code for the lock in Figure 3.4:

```
INT lock,       -- shared, equal to free/busy, modified by a client
    nowserving: -- shared, modified by SMserver, read by clients
-- create P processes pid[d] competing for a lock
PLACED PAR d = 0 FOR P
  INT copy:
  BOOL active:
  SEQ | pid
    busrq[d]:= TRUE               -- want to use a bus
    active := TRUE
    WHILE active
      SEQ
        WHILE d <> nowserving     -- wait for your turn
          WAIT(1)                 -- (simulates a fair bus arbiter)
        PRI PAR
          SEQ | high              -- Load Locked(LL)
            fcpu[d] ! dr | rrsize -- lock request
            tocpu[d] ? copy       -- lock copy into a local cache
          SEQ | low
            SKIP
        IF
          (copy = lock) AND (copy = free)
            SEQ                 -- reaching the free lock first
              busrq[d] := FALSE
              lock := busy      -- successful Store Conditional (SC)
              SERV(critical)    -- duration of critical section
              lock := free      -- unlock
              SERV(noncritical) -- duration of noncritical section
              active := FALSE   -- termination of an outer WHILE loop
          (copy = lock) AND (copy = busy)
            SEQ                 -- someone was much quicker
              WHILE copy = lock -- valid local copy of a busy lock
                WAIT(1)         -- wait for invalidation
          (copy <> lock)        -- invalid copy, get the fresh one
            SKIP                -- someone was quicker,is still using
                                -- the lock or released it already
```

Fig. 3.4: The essential part of `Test-and-Test & Set` lock model

1. A valid copy of a free lock is received and `Store Conditional` instruction succeeds. The successful process performs a critical section and then unlocks the lock.

2. A valid copy of a busy lock is received. The process waits until the copy is invalidated.

3. The `Store Conditional` instruction fails since the local copy has already been invalidated. A fresh copy must be obtained.

The *ticket lock* [49] uses two shared variables (`ticket` and `nowserving`) and a single atomic primitive `Fetch & Increment`. Each process uses the primitive when it first reaches the lock operation to obtain its ticket number from a shared counter.

```
[P] INT lock:     -- shared, initialized to busy except lock[0] = free
PLACED PAR d = 0 FOR P        -- create P processes worker
  INT copy:
  BOOL active:
  SEQ | worker
    active := TRUE
    WHILE active
      SEQ
        -- read miss for lock[d], request a fresh copy
        fcpu[d] ! dr | drsize
        tocpu[d] ? copy       -- lock[d] in a local cache
        IF
          copy = free         -- it is my turn
            SEQ
              SERV(critical) -- duration of CS in CPU cycles
              lock[d]:=busy
              lock[(d+1)REM P] := free   -- next process can go to CS
              active := FALSE            -- I have finished this time
          TRUE
            WHILE copy=lock[d]           -- correct copy,
              WAIT(1)        -- wait for invalidation in a local cache
```

Fig. 3.5: Array lock — Transim code

Then it busy-waits for the `nowserving` to reach the ticket number. The release method is to increment `nowserving`. Performance of the algorithm is similar to simple locks but its key advantage is fairness since processes are serviced in the order of their requests. On the contrary, simple locks do not provide any means for maintaining order of requesters and may therefore cause starvation of some processes.

The *array lock* [29] further reduces synchronization overhead since only one process incurs a read miss when a lock is released. If there are P processes that might possibly compete for a lock, then the lock data structure contains an array of P locations that processes can spin on, ideally each on a separate memory block to avoid false sharing. A process first uses a `Fetch&Increment` operation to obtain the next available location in this array (with wraparound) and then spins on this location. A process releasing a lock writes a value denoting *unlocked* to the next location in the array (after the one that the releasing process was itself spinning on). Only the process that was spinning on that next location has its cache block invalidated at the release. A sample Transim code for the array lock is given in Figure 3.5.

3.2.2 Barriers

Barriers are higher-level synchronization primitives often used in iterative algorithms. They are implemented with locks and additional shared variables and counters. Only barriers suitable for repeated use in loops are of interest.

```
-- local variable = shared flag at this point in each process
NOTE(reaching.barrier) -- message to simulation report
barcnt := barcnt+1
IF
  barcnt=P
    SEQ
      -- In timeless execution the process cannot be suspended,
      -- and therefore update of shared flag is safe.
      flag:= 1-flag    -- this makes local <> flag
      barcnt := 0
  TRUE
    SKIP
WHILE local=flag
  WAIT(1)
-- Processes waiting for the last one are released here.
local := 1-local       -- makes local=flag again
NOTE(leaving.barrier) -- message to simulation report
WAIT(1)    -- deschdules this process, so other processes can also
           -- see local=flag and leave the WHILE loop
-- Barrier overhead model may be put here
-- (a time delay as a function of P).
```

Fig. 3.6: Transim code for a sense-reversal barrier with an explicit overhead model

Centralized Barrier

A simple type of barrier is a *centralized sense-reversal* barrier [29]. Its implementation uses only three shared variables (`lock`, `flag` and `counter`) and performs synchronization of P processes in P steps. An implementation of the barrier's model is shown in Figure 3.6.

To obtain better accuracy of simulation in some cases, it is sometimes advantageous to use such a simple timeless barrier model with no overhead and insert an explicit overhead model (obtained by measurement) in front or behind it. The barrier overhead can mostly be approximated by a logarithmic or a polynomial function of P. The model uses only two shared variables (`barcnt` and `flag`) and one local variable named `local`. As noted in the code in Figure 3.6, a simulated process is descheduled only when timing statements (`SERV` or `WAIT`) are executed. Therefore atomic updates of shared variables can be easily modeled without a need of any special constructs.

Butterfly and Dissemination Barrier

To avoid contention for the same `lock` and `flag` variables by all processors in a centralized barrier, *butterfly* and *dissemination* barriers use software combining trees for synchronization [49]. Both barriers perform synchronization in $\lceil \log P \rceil$ steps and achieve better performance than a centralized barrier especially on systems with distributed networks and multiple parallel paths. However, since $O(P)$ bus transactions are still required, performance on a bus-based system may be similar to a centralized barrier because the bus transactions are serialized.

```
[P] INT counters:       -- shared array of counters
PLACED PAR d = 0 FOR P
  INT stage,wait.partner,data:
  SEQ | worker
    counters[d] := 0
    SEQ round = 0 FOR 3 -- barrier operation 3 times in a loop
      SEQ
        SERV(RAND(50,200))  -- random duration of tasks
        NOTE(entering.barrier)
        stage := 1         -- powers of 2
        WHILE stage < P    -- repeat in log P stages
          SEQ
            fcpu[d] ! dr | rrsize
            tocpu[d] ? data
            counters[d] := counters[d] + 1
            fcpu[d] ! dr | rrsize
            tocpu[d] ? data
            wait.partner := (P+d-stage) REM P -- partner to wait for
            WHILE counters[wait.partner] < counters[d]
              WAIT(1)
            stage := stage * 2 -- next power of 2
```

Fig. 3.7: A dissemination barrier — Transim code

Both barrier algorithms synchronize processors in pairs: each processor synchronizes with a partner at distance 2^{s-1} in step s. In a butterfly barrier, the two partners synchronize mutually and the algorithm therefore requires a processor count to be a power of two ($P = 2^N$) since $\frac{P}{2} \log P$ distinct pairs must be arranged in total. On the other hand, a dissemination barrier supports any number of processors: each processor sets an arrival flag of a partner to its right (modulo P) and waits for, then clears, its own arrival flag. The code of a dissemination barrier model is given in Figure 3.7.

3.2.3 Simulation of Synchronization Performance

It is not easy to assess and evaluate various types of locks with regard to overhead, bus contention and scalability. Locks are used for exclusive access to critical code sections (CS), thus enforcing sequential processing by processors. If a CS is in a loop, then fairness means, that only after all processes have acquired a lock in the first iteration, will processes start acquiring the lock second time, etc. The time needed to obtain a lock k-times by all P processes will therefore always contain a component $k * P * T_{\text{CS}}$. Time of waiting for the access to CS may be more or less overlapped with processing of non-critical code sections, so that the resulting performance or overhead are application-dependent. It is therefore questionable if duration of all critical sections and non-critical sections should be subtracted from the total execution time to obtain an overhead as described in [29].

Instead, we have measured the total communication time, or the bus occupancy time, Figure 3.8, which is application-independent. This bus communication may

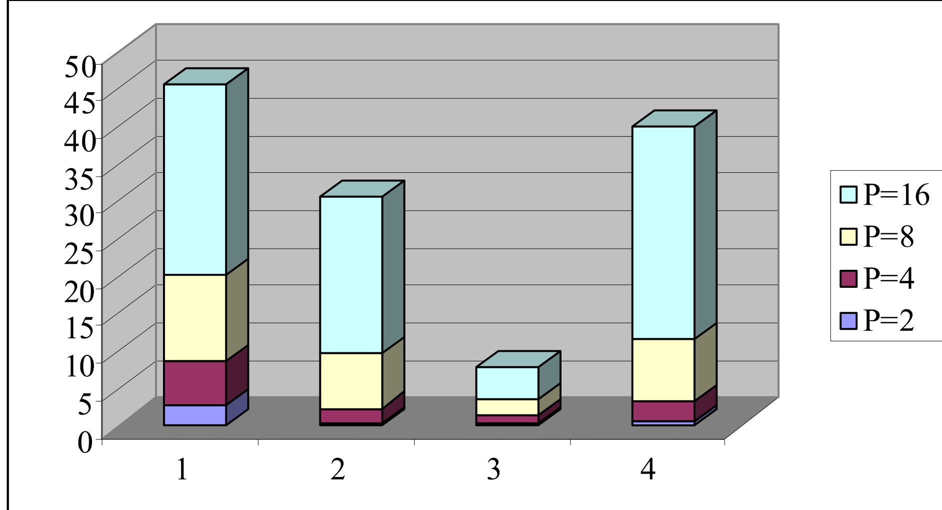


Fig. 3.8: A global time [μs] of bus transfers for various locks: 1) Test & Set, 2) Test-and-Test & Set), 3) Array lock, 4) Ticket lock

go on in parallel with useful work of other processors, using data in local caches. Nevertheless, during this work occasional read or write misses in shared variables other than lock variables may occur and related data requests may have to wait for free slots on a bus.

Bus busy times in Figure 3.8 have been measured on Transim models of locks with $k = 3$, $T_{\mathrm{CS}} = 100$ clock cycles ($= 0.5\,\mu$s), $T_{\mathrm{nonCS}} = 0$. The bus bandwidth was set to 5 000 Mbit/s what corresponds to a bus clock rate 50 MHz for 100-bit wide bus (32 address bits, 64 data bits). The results show the smallest bus traffic for the array lock and no advantage in performance of the ticket lock in comparison to the `Test-and-Test & Set` lock. For the simple `Test & Set` lock the bus utilization is close to 100 %.

Barrier overhead, as shown in Figure 3.9, is easily measured if the duration of all processes is exactly the same (argument of `SERV` in Figure 3.7). If we then subtract the total useful computation time (a number of loop iterations times T_{CS}) from the processing time, we will get the overhead. According to the simulation the dissemination barrier performs better than a simple counter-based barrier.

3.3 Parallel Benchmark Program

3.3.1 Algorithm Description

Simulation and performance prediction of bus-based shared memory multiprocessor will be demonstrated on the problem of computing the one-dimensional N-point discrete Fourier transform (DFT) on P processors. The sequential time complexity of an efficient Fast Fourier Transform (FFT) algorithm is known to be $O(N \log N)$ [50].

There have been several parallel FFT algorithms developed, which are suitable for different architectures. For example the *binary exchange algorithm* by Gupta and Kumar [51] uses a message passing pattern which is efficient especially on hy-

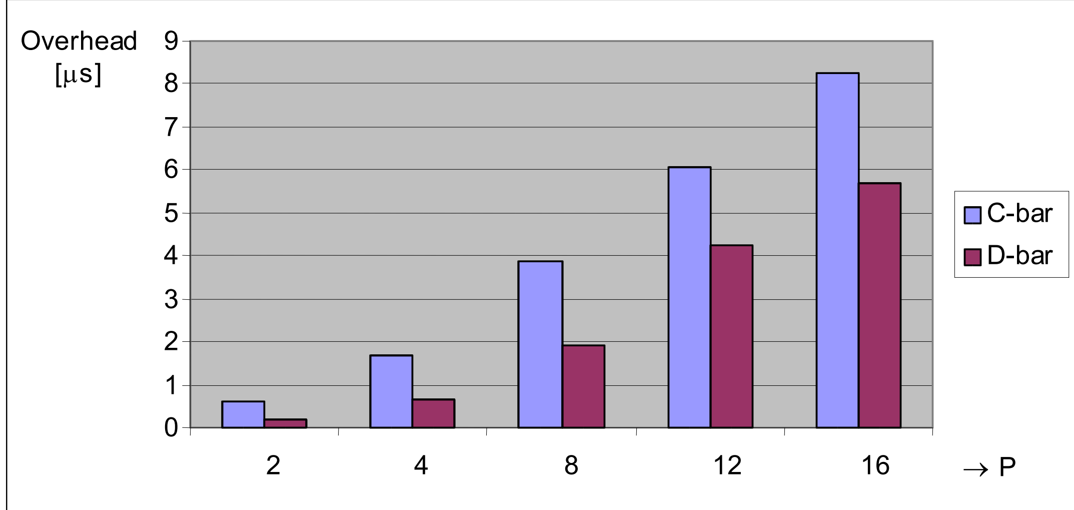


Fig. 3.9: Barrier-related overhead for a dissemination barrier (D-bar) and a counter-based barrier (C-bar)

percubes. Further we will consider an algorithm that reduces communication to a single transpose operation and is suitable for shared memory machines [52].

Let P divides N, $N = 2^q$ is a power of two and $N \geq P^2$. Let the N-element input vector $\vec{x}$ be represented by a matrix $X\left[\frac{N}{P} \times P\right]$ in row-major order. One column ($\frac{N}{P}$ elements) of the matrix X is first assigned to each processor. The algorithm is then performed in the following three stages:

1. The first stage involves a local computation of a FFT of size $\frac{N}{P}$ in each processor.

2. The second stage is a communication step that involves a transposition of the matrix with intermediate results.

3. Finally, $\frac{N}{P^2}$ local FFTs, each of size P, are sufficient to complete the overall FFT computations on N points.

The amount of computation work done by P processors for the FFT of an N-element real vector is $\frac{N}{2}\log N$ "butterfly" operations, where one butterfly represents four multiplications and six additions/subtractions (twenty CPU clocks in simulation). The work is divided into stage one and three (equation 3.1).

$$P\left[\frac{N}{2P}\log\frac{N}{P} + \frac{N}{P^2}\frac{P}{2}\log P\right] = \frac{N}{2}\log N \tag{3.1}$$

Parallelization of FFT is illustrated in Figure 3.10 for $P = 4$ and $N = 16$, with the same work done in stage one and three. Each processor does one 4-point FFT before a communication step and one 4-point FFT after it. Typically, however, the work done in stage one proportional to $(\log N - \log P)$ will be much larger than the work done in stage three, which is proportional to $\log P$. The only overhead in parallel implementation is due to a matrix transposition. The matrix transposition problem requires $P(P-1)$ bus transactions on a bus.

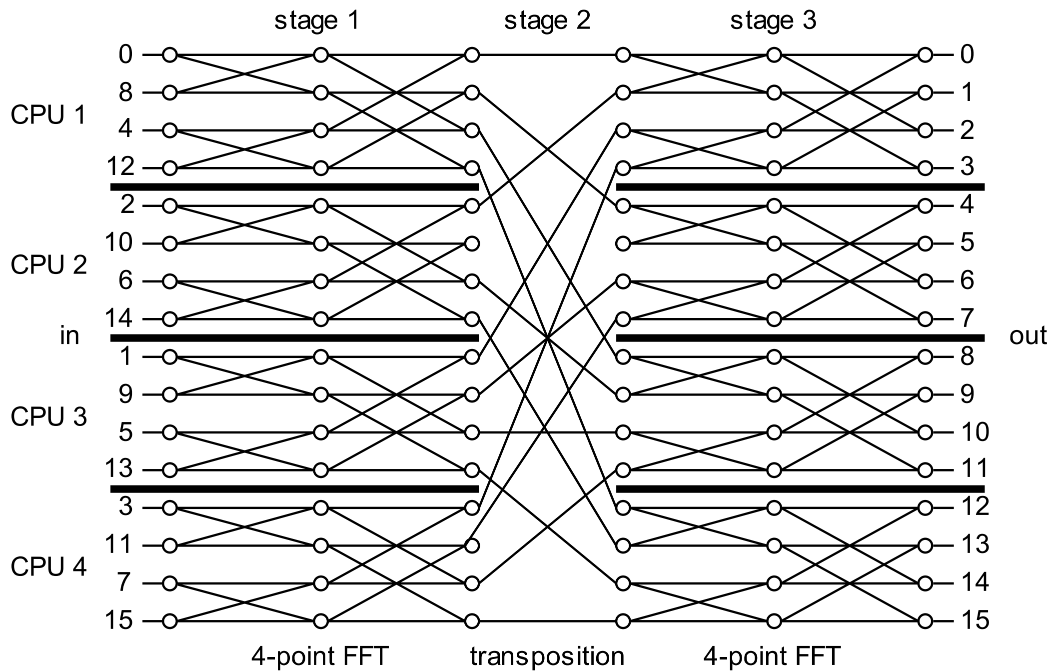


Fig. 3.10: Parallel FFT for $N = 16$ and $P = 4$

Processors write the results of the $\frac{N}{P}$-point FFT computed in stage one into the local caches and do the transposition at the same time. This means that consecutive values of FFT will be stored with a stride required by the rule of matrix transposition. The following read requests by other processors at the start of stage three will generate read misses: at cache block size 16 bytes, one miss always after three hits in a sequence. Fresh cache blocks will be loaded into requester's cache and simultaneously into the shared memory.

FFT processing is continuous in real time. Therefore loading of the next input vector from outside and writing the previous results from processors to environment is carried out in the background, in parallel with the first stage of processing, using double buffering scheme.

3.3.2 OpenMP Implementation

To test accuracy of simulation, an implementation of the described FFT algorithm has been developed and executed on Sun Enterprise 450 server with 4 Sun UltraSPARC II/400 CPUs at 400 MHz. The program was implemented in C with OpenMP directives used for specification of parallel execution.

Parallelization of loops, the way as it is done in OpenMP [12], uses basically three loop scheduling types, static, dynamic and guided scheduling. Their simulation in Transim is straightforward. For example the guided scheduling is illustrated in Figure 3.11. Initialization at the beginning of the code is done by the process that has arrived to this point first, other processes skip over it. This kind of condition synchronization (Eureka) is based on a shared integer flag.

```
-- Guided Loop Scheduling
IF
  flag = thispoint            -- first CPU at this point does
    SEQ
      rest := itercount       -- initialization
      flag := flag + 1
  TRUE                        -- other CPUs skip over
    SKIP
WHILE rest > 0
  SEQ
    IF
      rest > P
        mychunk := rest/P
      TRUE
        mychunk := 1
    base := base + mychunk
    rest := rest - mychunk
    SEQ h = 0 FOR mychunk     -- process your chunk of iterations
      SERV(RAND(low, high))   -- iterations of different duration
```

Fig. 3.11: The code for simulation of guided loop scheduling

3.3.3 Simulation Model Parameters

Performance prediction gives an estimate how many FFT operations can be done per second. The simulated SMP has had the following features and parameters:

- an atomic bus
- fair bus arbitration policy
- 100 MB/s bus bandwidth
- 50 MHz bus clock
- a miss penalty of 20 CPU clocks
- a L1 cache block size 16 bytes.

The size of the cache is assumed to be sufficient to hold relevant segments of input data (a real vector), intermediate data after the first stage of FFT (a complex vector) as well as the results (a complex vector). In the worst case ($P = 2$) the size of all these vectors will be around 10 KB, if we use `REAL32` format. We assume I/O connected via a bus adapter directly to the cache. To avoid arbitration between CPU and I/O, the next input and previous results are transferred in/out during the first stage of the FFT algorithm. E.g. for $N = 1024$, $P = 4$, and CPU clock speed 400 MHz, the simulation gives a duration of one FFT operation 759 μs.

Although only a single level of cache was simulated, the results presented in section 3.4 indicate that this abstraction is sufficient to obtain reasonable accuracy.

N	2048	1024	512	256	128	64	32
T **[ms] (Transim)**	1.445	0.759	0.424	0.261	0.181	0.142	0.123
T **[ms] (Sun)**	1.431	0.733	0.403	0.242	0.172	0.139	0.121
Difference [%]	+1.0	+3.5	+5.2	+7.9	+5.2	+2.2	+1.7

Tab. 3.1: Results of real and simulated 1D N-point parallel FFT on 4 processors

Consideration of more levels of cache makes the model much more complex without a significant improvement of accuracy, which could justify the complexity

Four barriers, some of them implicit in OpenMP directives, have been inserted in Transim code. A data prefetch has not been supported by hardware and therefore not considered in simulation. If it was available, it could improve performance further.

3.4 Results And Conclusions

The model of SM architectures and synchronization primitives based on message passing in the CSP style proved to be a valuable means of performance prediction. The description of hardware architecture, software and mapping to one another in Transim tool is very concise and directly executable. As for the accuracy of performance prediction, it was better than 8 % at the parallel FFT benchmark in most cases. A Transim code had some 480 lines and simulation executed on the same server as the real program took up similar times like real execution. The results of real and simulated computations of 1D N-point parallel FFT on 4 processors as published in [68] are given in Table 3.1.

Other algorithms have also been successfully simulated using the described models of shared memory architectures and synchronization primitives. Results obtained from simulation of a sorting algorithm PSRS (Parallel Sorting with Regular Sampling) can be found in [64]. Simulated solution of large systems of linear equations on various architectures including shared memory multiprocessors is described in [67].

Chapter 4

Clusters of Workstations and SMPs

4.1 Cluster Interconnection Models

Clusters are usually built of commodity hardware: personal computers or workstations and a standard communication infrastructure commonly used in local area networks (Ethernet, Token Ring, etc.). Dedicated clusters often use special-purpose interconnections like Myrinet [8] designed especially to obtain low communication latency.

A typical topology of an interconnection network used with slow 10 Mbit/s Ethernet was a bus. Due to its low bandwidth and shared communication medium, this architecture is unsuitable for high performance computing and has already been overcome by alternatives with higher performance. Nevertheless, simulation of a bus interconnection can be accomplished as well by adapting the model described in chapter 3.

The most common interconnections in today's clusters are switched networks with a star topology. Computers in such networks are directly connected to a high speed hardware router usually by 100 Mbit/s or 1 Gbit/s Ethernet links. Since internal structure of routers varies, several models have been developed and are described in the following sections.

4.1.1 Omega Network

Omega network is an example of a blocking multistage indirect network which has n inputs and n outputs. There are several equivalent (isomorphic) networks (Omega, Butterfly, Hypercubical, Baseline) that differ in permutation connection between stages [54]. Omega network with $n = 8$ is depicted in Figure 4.1. The figure shows a topology of the network and a process graph with channel connection of a Transim model of one of its switch modules. The network has $\log_2 n$ stages. Each stage consists of $n/2$ switch modules. A single switch module is usually connected to two input channels and two output channels (2×2 switches), even though $k \times k$ switches are possible as well. Interconnection between stages uses a *perfect shuffle* permutation.

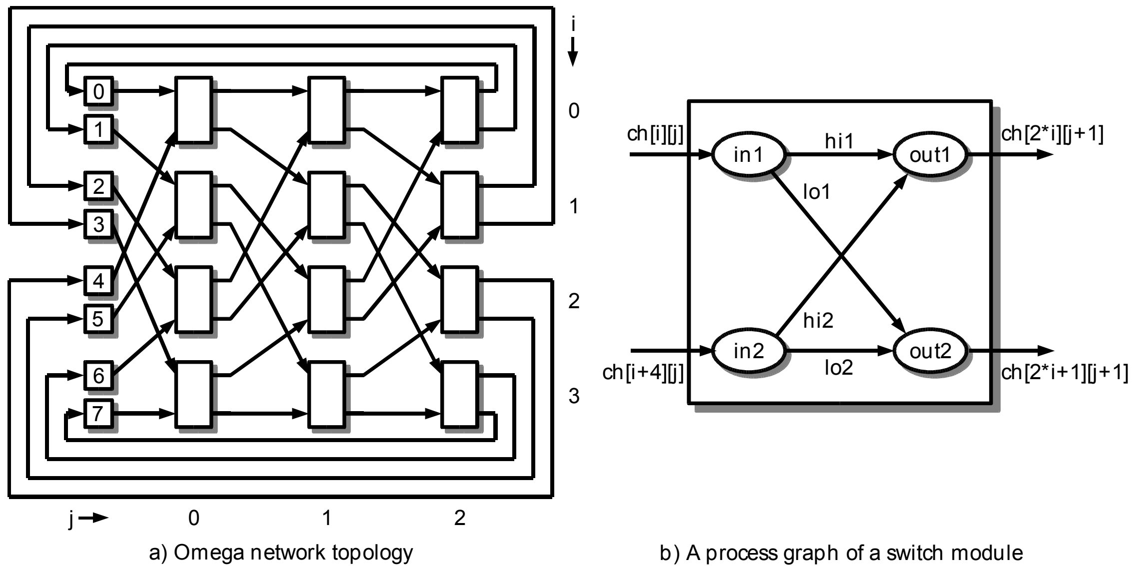


Fig. 4.1: Omega network topology and a process graph of one of its switch modules

```
INT position,t,x:
SEQ | in1                 -- upper input, j is a stage index
  IF                      -- creating masks for 3 bits of the header:
    j = 0
      position := 4       -- 2^2, MSB of dst address
    j = 1
      position := 2       -- 2^1
    j = 2
      position := 1       -- 2^0, LSB of dst address
  WHILE TRUE
    SEQ
      ch[i][j] ? x        -- wait for INT on ch[i][j] and put it in x
      t := (x/position) REM 2  -- j-th address bit
      IF
        t = 0
          hi1 ! x | size   -- to upper output
        t = 1
          lo1 ! x | size   -- to lower output
```

Fig. 4.2: Transim code of an input process running in an Omega network's switch module

The switch receives messages from its input channels and passes every received message to one of its output channels (or possibly to all of the outputs if it supports broadcast). Routing of messages is based on a binary representation of a destination address. A different bit of the address is used in each stage to determine if a message should be routed to an upper or lower output channel of a current switch.

A developed Transim model uses four parallel processes to model a single switch module in the network as shown in Figure 4.1 b). Two of the processes receive messages from the input channels and send them to one of the two output processes,

```
INT a:
SEQ | out1
  WHILE TRUE
    SEQ
      ALT              -- wait for data on channel hi1 or hi2
        hi1 ? a
          SKIP
        hi2 ? a
          SKIP
      ch[2*i][j+1] ! a | size    -- send it via upper output
```

Fig. 4.3: Transim code of an output process running in an Omega network's switch module

which in turn passes the messages to the next stage of the network. Since only a single integer can be communicated between processes in the Transim simulator, it is used to pass both a destination address (3 bits in case of 8×8 Omega network) and a length of the message (the remaining bits). One of the bits can also be used to indicate a broadcast operation.

A code of one input process (denoted `in1`) is shown in Figure 4.2. The other input process is very similar. An output process (denoted `out1`) is even simpler, because it does not do routing. Instead, it only passes on all messages received from the input processes. The code is given in Figure 4.3. Variable `i` is an index of a particular switch module in a given stage determined by index `j`.

4.1.2 Fat Tree Topology

A fat tree is another kind of a blocking multistage indirect network. Similarly to Omega networks it is built of simple switch modules, but input/output links are bidirectional. One way of creating fat trees is to put two unidirectional multistage networks back-to-back and then fold one onto another. Communication between two partners requires almost twice as many hops than the unidirectional network, but it provides multiple paths between any two nodes.

A fat tree achieves better performance over a simple tree due to increased bandwidth in higher levels. Links used for transferring a message from lower levels are selected pseudo-randomly. A structure of a fat tree connecting eight nodes by six switches is shown in Figure 4.4. Each switch is connected to four bidirectional links.

Since all four links connected to each switch module are bidirectional, four input processes and four output processes are used to simulate functionality of a switch. It is not possible to use only a single input process which would receive from multiple input channels by using `ALT` statement. Such an implementation may lead to a so called *fetch deadlock* (at least two processes execute pending send operation and cannot receive). More information on deadlock-free routing can be found in section 5.1.

The input processes receive messages and pass them to appropriate output processes according to destination addresses of the messages. The integer communicated

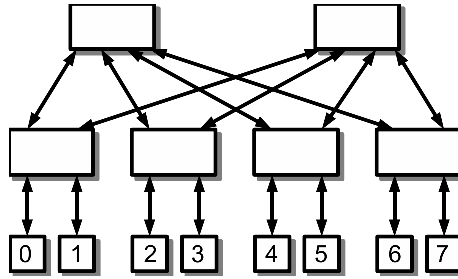


Fig. 4.4: Fat tree topology

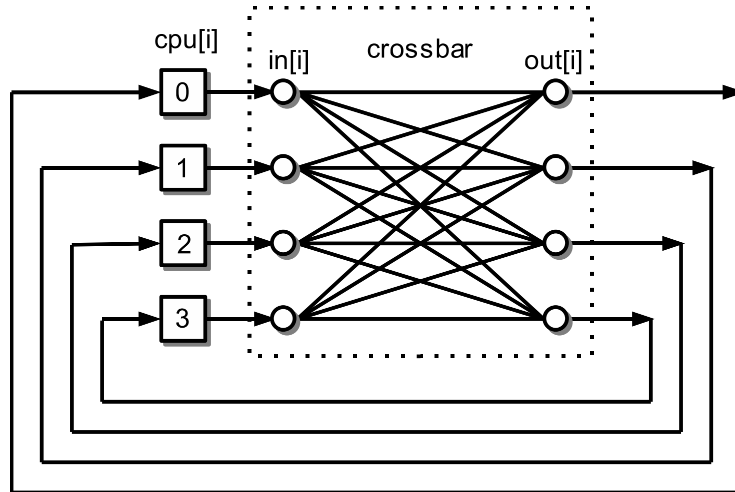


Fig. 4.5: Process graph of a crossbar switch model

in a simulation is again used to store both an index of a destination processor and a length of the message.

4.1.3 A Crossbar Switch

A crossbar switch is one of the most complex nonblocking networks and therefore it can achieve very good performance. It is built of $m \times n$ simple crosspoint switches, which enable m concurrent data transfers to $m < n$ different nodes. A common configuration is a $n \times n$ square crossbar, which can implement any of $n!$ permutations without blocking.

If multiple source nodes send messages to the same destination at the same time, only one of the requests is served at a time. The order in which requests are served is determined by an arbitration logic (priority, fairness, random, etc.). In case of simulation, arbitration is provided by `ALT` statement in processes `out[i]`. A particular arbitration strategy is simulated by variable indexing on input channels of the `ALT` statement. The principle is analogous to simulation of a bus arbiter as described in section 3.1.1 on page 26.

A model of a 4×4 crossbar switch is depicted in Figure 4.5. Each of the four input processes (denoted `in[i]`) receives messages from an appropriate node `cpu[i]`, decodes a destination address and passes messages on to one or more of the output processes `out[i]`, which are implemented using `ALT` statement.

A more complex model using a different arbitration approach has also been tested. Instead of distributed arbitration a central arbiter is used, which receives requests for communication from senders and receivers. After the arbiter assigns a communication channel to a sender-receiver pair, it sends the channel's index to both partners and they start communication. Such an approach allows simulation of complex arbitration strategies.

4.1.4 Models of Switching Techniques

Interconnection networks use various switching techniques to pass data between their stages. Some of the possible strategies are as follows:

- **Circuit switching** (CS) is a technique adopted from telephone networks. The whole path from a source to a destination is constructed before a transmission of data starts and it is reserved until all data have been transmitted.

- **Store-and-forward** (SF, also called packet switching) divides a message into packets of a certain size, which fit into buffers of switches. Received packets are *individually* routed to their destination. An important aspect is that every packet is passed on to another switch only after all of its contents have been received.

- **Virtual cut through** (VCT) also divides messages into packets but each packet is subdivided into flits. Unlike the store-and-forward technique, flits can be passed on just after a header flit has been received.

- **Wormhole switching** (WH) is very similar to virtual cut through but it uses switches with more limited buffers, which cannot hold a whole received packet. Due to this limitation the technique is prone to blocking and deadlock.

The models of interconnection architectures described so far can be modified to support modeling of various kinds of switching. For example congestion at group communications (simultaneous communication requests on multiple channels) are resolved by serialization (selection statement `ALT` in a loop) for store-and-forward routing or by interleaving flits (bytes) from different messages for wormhole routing. In the latter case a longer message is sent as a sequence of individual bytes using zero start-up time (parameter `ECD=0` of the used channel) and an explicit `WAIT(start-up)` statement.

Moreover, the models can support different communication latencies for various source-destination pairs. This occurs when various communication paths go through a different number of smaller crossbars within the interconnection network, e.g. Myricom's Myrinet scalable network created by interconnection of 8×8 crossbars in a fat-tree topology as described above. In the highest level of detail, each smallest building block, a 2×2 crossbar, can be modeled individually,

4.2 SMP Cluster Model

In the last few years, clusters of symmetric multiprocessors (SMPs) have gained significant popularity in the field of parallel computing due to their excellent price/performance ratio and possible universal use. The price of PCs built with two processors is not very different from PCs with a single processor only, so it is common to have multiprocessor servers and even multiprocessor personal PCs.

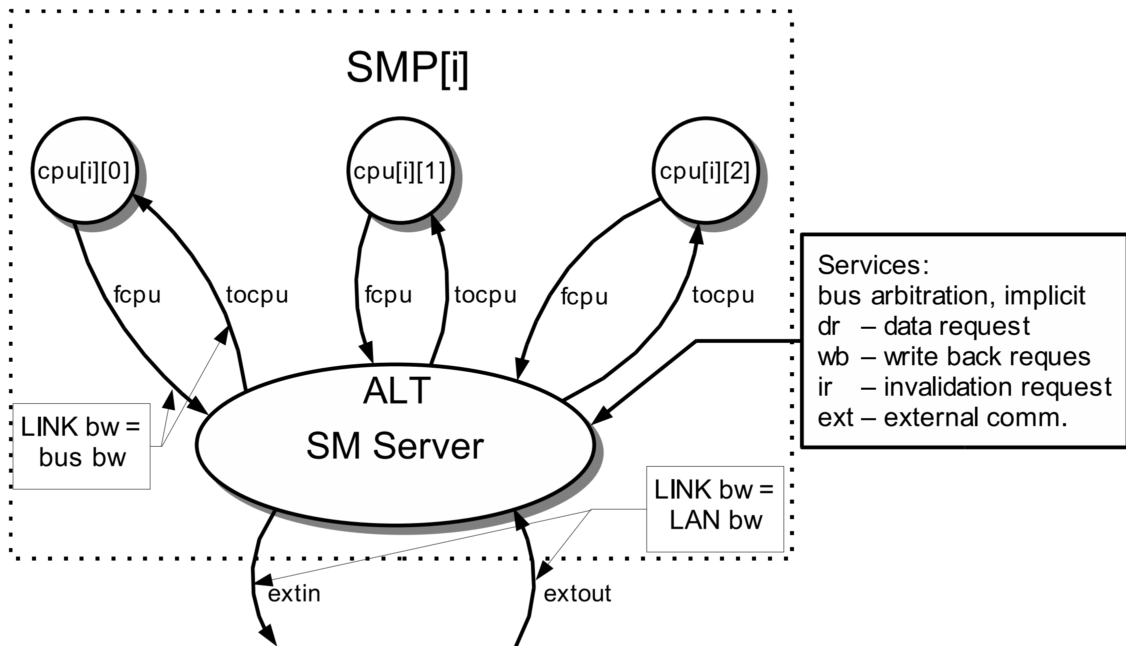
As in the case of single processor PCs, there is a huge effort to use this computation power in clusters of PCs interconnected by general-purpose networks or by specialized communication hardware. However, it is not easy to decide whether a cluster of PC-based SMPs is a better choice than expensive and dedicated SMP machines with many processors. The overall performance is influenced by many factors, especially by a particular cluster configuration, internal communication within SMP nodes and external communication among SMP nodes themselves.

A model of a bus-based symmetric multiprocessor has already been described in chapter 3. A cluster of single-processor PCs can be easily modeled by using one of the described switch models (section 4.1) for passing messages among connected nodes. The two models can also be merged to produce a more complex model of a cluster of SMPs, which can help investigation of various architectures and aid making decisions about the most suitable configuration for a particular algorithm.

Simulation of a hybrid architecture of SMP clusters in Transim as described in [66] uses a feature of the simulator language which supports setting different speeds and delays on various inter-processor channels. This enables setting appropriate parameters for both internal links in SMP nodes as well as external ones among SMP nodes. Transim allows different speeds on interconnecting links if topology is described by `LINK` statements (see `LINK` bandwidth assignments in Figure 4.6). Otherwise, in the case of uniform speed, topology information is extracted from the simulation file automatically.

A process graph of a SMP cluster model with four SMPs and three processors per SMP is depicted in Figure 4.6. Top of the figure shows a graph of a single SMP node, which is an extension of the model described in chapter 3. In addition to the original model, a `SMserver` process supports one more service (`ext` — external communication request) and the process is also connected to two more channels (`extin` and `extout`), which are used for communication with other nodes. Internal communication of processes `cpu` in a single SMP node is managed by the `SMserver` process.

Bottom of Figure 4.6 shows a connection of SMP nodes by a crossbar switch. When a process `in[i]` receives data from a node `SMP[i]`, it either passes the data to one of the processes `out[j]`, $i \neq j$, in case of a point to point communication, or the process `in[i]` splits into three processes (`a`, `b` and `c`) which pass the data to all the other processes `out[j]` in parallel in case of a broadcast communication. Each process `out[j]` passes all data received from processes `in[i]` to its node `SMP[j]`.



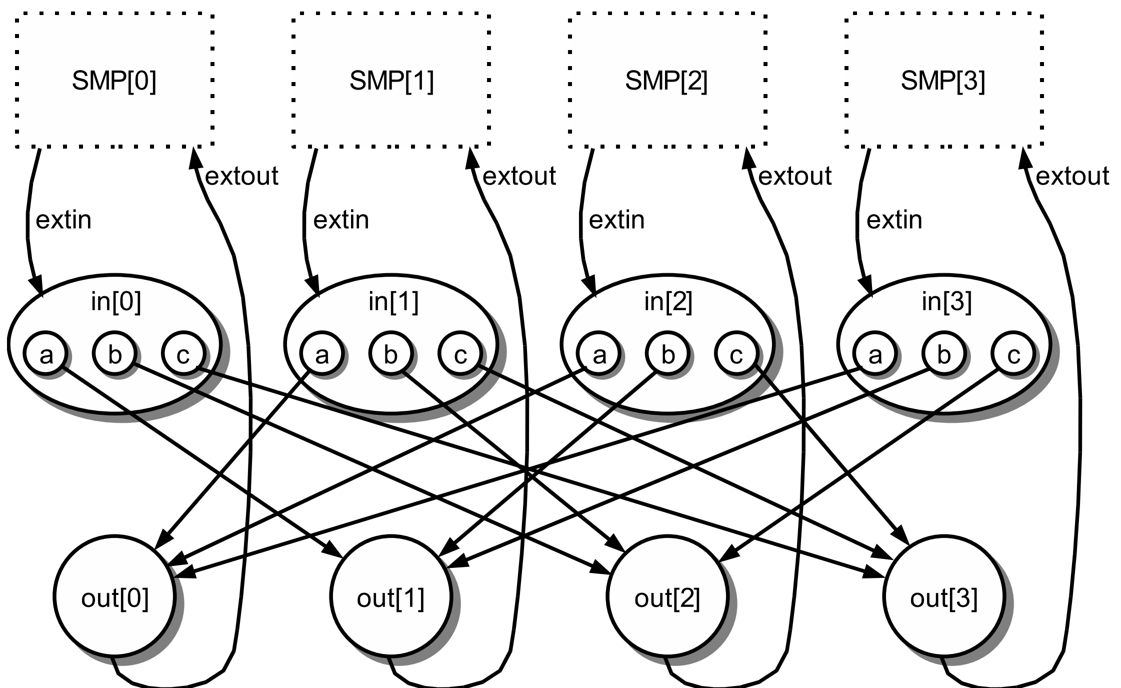


Fig. 4.6: Model of a cluster of SMP nodes connected by a crossbar switch — a process graph

4.3 A System of Linear Equations Benchmark

The described models were tested by simulation of a parallel version of the Gauss-Jordan elimination method for solution of linear equations. In this method the algebraic system is transformed directly to a diagonal form and there is no back substitution phase. The main part of the diagonalization process of the Gauss-Jordan method can be described by the following pseudo code (A is a square matrix of size $n \times n$, $\vec{b}$ is a vector of right-hand sides of size n):

```
for i := 1 to n do                          pivot row
  for j := 1 to n, (j ≠ i) do               for all other rows
    begin
      c = a_ji / a_ii
      b_j := b_j − c b_i
      for k := i + 1 to n do                for each element in current row
        a_jk := a_jk − c a_ik
    end
```

This simple version of the algorithm supposes that a value of the element a_{ii} is never zero or near zero. In a case when zeros on the main diagonal may occur, a so called *partial pivoting* modification would have to be used. This method selects a pivot row depending on absolute values of elements on the main diagonal.

The algorithm is further illustrated by Figures 4.7 and 4.8. Figure 4.7 shows elements of the matrix A with the vector $\vec{b}$ added as the last column in four different steps of the computation:

a) at the beginning of the algorithm

b) after the first pivot row was divided by an element on the main diagonal

c) after the first pivot row was subtracted from other rows

d) a final unit matrix with a vector of solutions $\vec{c}$

The process is also schematically shown in Figure 4.8. Black squares represent ones, white squares represent zeros and gray squares represent any values.

A parallel version of the Gauss-Jordan method for P processors assigns each processor P/n consecutive rows of a system matrix A and P/n elements of vector $\vec{b}$ for storage and processing. Each processor takes its turn as a "leader" and sequentially broadcasts each one of its P/n equations (a pivot row) and after each broadcast it also modifies its remaining equations.

4.3.1 Simulation of a Workstation Cluster

To be able to compare results of the cluster model simulations and real executions, the Gauss-Jordan algorithm was implemented and executed on a cluster of eight Ultra 5 workstations with Ultra-SPARC II processors running at 270 MHz. Each

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{pmatrix} \qquad \begin{pmatrix} 1 & \frac{a_{12}}{a_{11}} & \cdots & \frac{a_{1n}}{a_{11}} & \frac{b_1}{a_{11}} \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{pmatrix}$$

a) original matrix b) the 1st pivot row modified

$$\begin{pmatrix} 1 & \frac{a_{12}}{a_{11}} & \cdots & \frac{a_{1n}}{a_{11}} & \frac{b_1}{a_{11}} \\ 0 & a_{22} - a_{21}\frac{a_{12}}{a_{11}} & \cdots & a_{2n} - a_{21}\frac{a_{1n}}{a_{11}} & b_2 - a_{21}\frac{b_1}{a_{11}} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & a_{n2} - a_{n1}\frac{a_{12}}{a_{11}} & \cdots & a_{nn} - a_{n1}\frac{a_{1n}}{a_{11}} & b_n - a_{n1}\frac{b_1}{a_{11}} \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & \cdots & 0 & c_1 \\ 0 & 1 & \cdots & 0 & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & c_n \end{pmatrix}$$

c) other rows modified by the 1st pivot row d) solution matrix

Fig. 4.7: A matrix of an algebraic system in four phases of Gauss-Jordan elimination algorithm

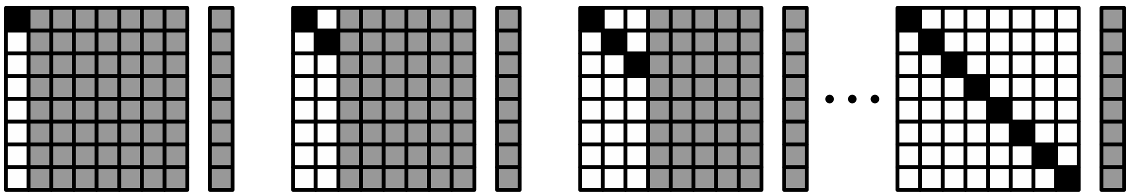


Fig. 4.8: Four different states of matrix A during computation of Gauss-Jordan elimination method

node's line card was connected to a hardware multiport router Summit48, which has 48 Ethernet ports at 10/100 Mbit/s and two Ethernet ports at 1Ġbit/s. The router has a 17.5 Gbit/s non-blocking switch fabric and a forwarding rate of 10.1 million packets per second. Hardware latencies of the router are below 10 μs and can be neglected with respect to the software latencies (160 μs for send and 260 μs for receive operation at clock speed 270 MHz).

The program was implemented in C with Message Passing Interface (MPI) library used for parallel execution and communication. The algorithm requires only one type of a collective communication function — broadcast, which is used for sending pivot rows to all processors.

Parameters of the simulation model were set to match the real environment as close as possible. Interesting parameters of processing elements simulating the crossbar switch and workstations configurable in Transim are clock speed (`SPD`), link speed (`LS`), external channel delay (`ECD`) and internal channel delay (`ICD`). The parameters were set so that the total simulated latency corresponds to the latency of a MPI broadcast routine, which in turn depends on a number of nodes involved in the communication. Measured values of broadcast latencies are about 500, 750 and 1000 μs for 2, 4 and 8 nodes whereas related transfer rates are 10, 5 and 3.5 Mbyte/s respectively.

Table 4.1 contains a comparison of speedup values obtained from simulations and from executions on a workstation cluster. A solution of systems of two hundred

	2 processors		4 processors		8 processors	
# equations	S_{sim}	S_{real}	S_{sim}	S_{real}	S_{sim}	S_{real}
200	1.01	0.87	0.99	0.81	0.87	0.74
400	1.58	1.54	2.18	2.16	2.45	2.58
600	1.78	1.70	2.85	2.87	3.81	4.18
800	1.86	1.80	3.21	3.17	4.79	5.06
1000	1.90	1.86	3.42	3.34	5.47	5.53

Tab. 4.1: Speedup for simulated (S_{sim}) and real execution (S_{real}) of a linear equation benchmark program on a cluster of workstations

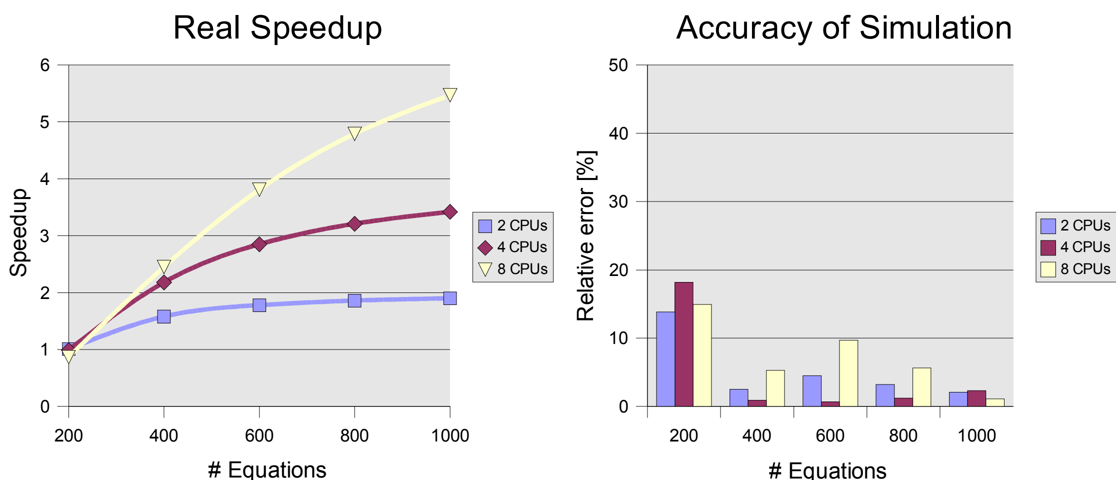


Fig. 4.9: Speedup of real execution of a linear equation benchmark program and achieved relative accuracy of simulation

to one thousand equations on two, four and eight processors was performed and simulated. The data are also plotted in Figure 4.9. Graphs in the Figure 4.9 show speedup of the real implementation and a relative difference between measured and simulated values of the speedup. The results indicate quite high accuracy of simulation (a difference within 10 % in most cases), which has been achieved by measuring durations of inner loops (sequential code) in a real program and using the values in simulation (an argument of the `SERV()` command).

4.3.2 SMP Cluster Simulation

A hybrid (message passing and shared variable) programming paradigm was simulated with the developed model of a SMP cluster. Combining two programming models is beneficial if granularity of communication is small for a shared memory and large for message passing. A simulated algorithm was the same linear equation solver described at the beginning of section 4.3.

Four SMP nodes, each with three processors on a shared bus, were interconnected through a 4×4 crossbar switch. Equations of the solved system are assumed equally

	SMP Cluster 4 × 3		Single SMP 1 × 12		CPU Cluster 12 × 1	
n	**Time [ms]**	**Speedup**	**Time [ms]**	**Speedup**	**Time [ms]**	**Speedup**
120	2.92	5.0	3.61	4.0	2.87	5.0
240	14.1	8.3	33.9	3.4	14.4	8.1
360	40.6	10.0	117	3.4	42.8	9.2
480	88.9	10.5	278	3.4	96.3	10.0
600	167	11.0	543	3.4	183	10.0
720	280	11.3	939	3.4	311	10.1
840	436	11.5	1491	3.4	489	10.3

Tab. 4.2: Execution times and speedup for simulated architectures with limited L1 cache size

distributed among processors. The leader processor broadcasts the pivot row to the remaining SMPs and then all processors transform all their equations accordingly. A broadcast within a single SMP is done by reading the new pivot row, previously modified by the leading processor, by other processors. Reading starts after a barrier synchronization at the end of each modification step. Barrier synchronization among SMPs is achieved in a simulation by a shared counter in each SMP that is set by the leader and tested by processors within each SMP.

Actual parameters of simulated architectures have been the following: L1 cache size: 16 KB with access time 1 CPU clock cycle, access time to external memory or L2 cache (of infinite size): 3 clock cycles, clock speed: 1 GHz in all three cases, bus bandwidth in SMPs: 1 GB/s, external channel speed: 100 MB/s, start-up time: 10 μs.

The results of simulations for a number of equations n varying from 120 to 840 and a processor count equal to 12 are in Table 4.2. The values of both simulated execution time and speedup are also plotted in Figure 4.10. The single SMP performs poorly due to bus saturation caused by frequent requests for data from processors. A cluster of a dozen of workstations shows a good speedup for more than a few hundred equations. The SMP cluster gives the best results (although it is comparable to the workstation cluster) since for three CPUs the SMP bus bandwidth is sufficient.

4.4 Conclusions

Simulation of a linear equation solver on a SMP or SMP cluster in Transim is very time-consuming, since essentially every cache miss is simulated. A single simulation can easily take several hours. Another restriction is that the simulator supports only about 2^{32} events after which simulation automatically terminates. Both these drawbacks can be eliminated by parallel simulation, which was successfully used to obtain presented results.

In fact, simulation of the linear equation solver is embarrassingly parallel application, since, contrary to real processing, to start up simulation in any given state,

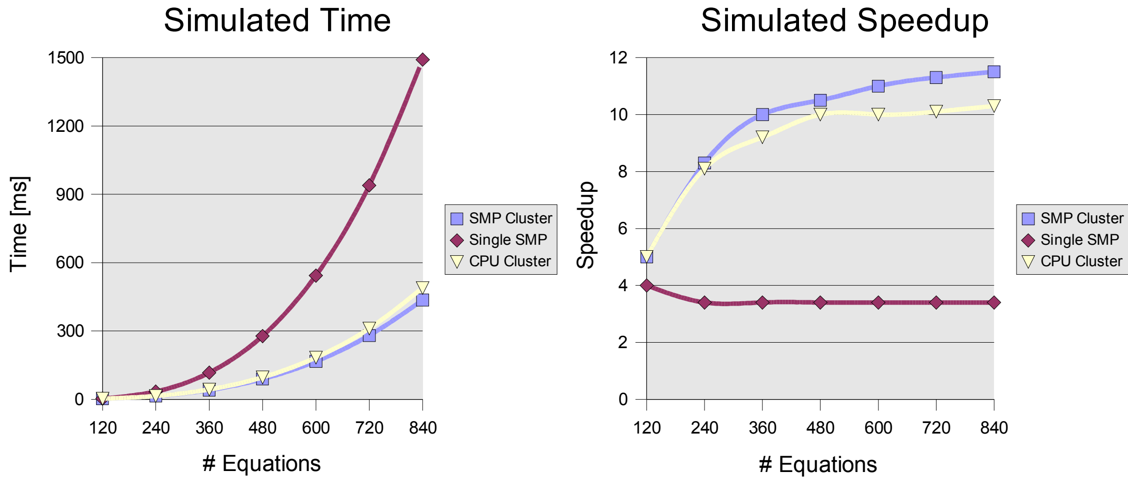


Fig. 4.10: Execution times and speedup for simulated architectures with limited L1 cache size

the value of only a single shared variable is needed — the number of equations yet to be transformed. Simulation can thus run on several workstations in parallel, each workstation starting simulation from a different pivot row.

To balance the load, the number of pivot rows in care of individual workstations should differ. The load is given by the total length of all pivot rows in a single workstation and this quantity should be approximately the same for all workstations in order to complete the simulation simultaneously. The overall average efficiency E is calculated from partial efficiencies figured out by workstations involved as a weighted average with weights given by simulated execution times. The global speedup is then $S = PE$ where P is the processor count in the simulated system.

Aside from the presented hybrid shared variable/message passing programming paradigm, simulations of other styles of programming were also performed with the model of a SMP cluster:

- process farm
- pipeline
- data parallel computing.

The model was configured with number of nodes and number of processors per a single node varying from one to sixteen. Three different configurations of node interconnect were used:

- optimized hardware configuration
- LAN configuration
- WAN configuration.

Results of the study published in [66] also indicate usefulness of the approach to support a qualified choice of the most suitable configuration for a particular speed of communication links and given tasks to be solved.

Chapter 5

Tuning the Performance of Communication Algorithms

Combinatorial search and optimization techniques in general are characterized by looking for a solution to a problem from among many potential solutions. For many search and optimization problems, exhaustive search is infeasible and some form of directed search is undertaken instead. In addition, rather than only the best (optimal) solution, a good non-optimal solution is often sought.

As with any kind of iterative algorithm, it is impossible to predict run time of search or optimization algorithms since it depends on input data, randomly generated numbers etc. Nevertheless simulation of such algorithms helps a developer to design an efficient parallel implementation by identifying potential bottlenecks, sources of overhead and influence of a selected granularity of the algorithm and a type of a target architecture. This chapter describes design and performance tuning of an iterative algorithm which optimizes scheduling of communications on irregular interconnection networks.

The particular optimization task has been chosen since communications between two partners (point-to-point) or among all (or a subset) of partners engaged in parallel processing have a dramatic impact on the speedup of parallel applications. Performance modeling and optimization of communications is therefore important in design of application specific systems.

An optimization part of the described algorithm is based on *genetic algorithm*, which is a powerful, domain-independent search technique that was inspired by Darwinian theory. It emulates the natural process of evolution to perform an efficient and systematic search of the solution space to progress toward the optimum. It is based on the theory of *natural selection* that assumes that individuals with certain characteristics are more able to survive and hence pass their characteristics to their offsprings. It is an *adaptive* learning heuristic belonging to a class of general *nondeterministic* algorithms.

Genetic algorithms operate on a *population* (or set) of *individuals* (or solutions) encoded as strings. These strings represent points in the search space. In each iteration, referred to as a generation, a new set of strings that represent solutions (called offsprings) is created by crossing some of the strings of the current generation. Occasionally new characteristics are injected to add diversity. Genetic algorithms

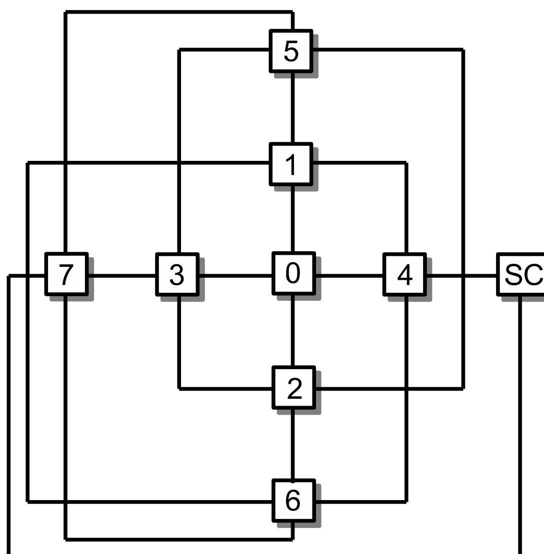


Fig. 5.1: Eight-processor AMP configuration

combine information exchange along with *survival of the fittest* among individuals to conduct the search. Convergence of genetic algorithms has been proved by use of Markov chains and a fundamental *Schema Theorem* [55, 56].

5.1 Models of Communications on Irregular Topologies

Processors in common distributed-memory parallel computers are often interconnected with networks that are node-symmetric, i.e. each node has the same view of the network. Such regular networks like a ring, a 2D-torus or a hypercube have the advantage that the same relatively simple routing function can be used, identical in (translated to) all nodes. All-to-all communications then require that all nodes communicate simultaneously without conflict, i.e. no channel can be used at any time in one direction by more than one message (e.g. the time-arc-disjoint trees — TADTs have this property).

However, there are cases when irregular networks have a certain advantage and are given priority over the regular networks. An example of such irregular network topology is the class of AMP (A Minimum Path) configurations, designed especially to minimize the network diameter and the average inter-processor distance [57]. The AMP networks have been found for node count $P = 5$, 8, 12, 13, 14, 32, 36, 40, 53, 64, 128, 256.

The 8-processor AMP network topology is depicted in Figure 5.1. The *SC* node denotes a system controller (host computer) that sends input data to processing nodes and collects results. Each processing node can communicate simultaneously on four bi-directional full duplex links.

There are a few routing techniques suitable for irregular networks such as interval routing or up-down routing [58]. However, implementation of group communication

algorithms is seldom mentioned in literature. Next paragraphs will therefore define group communication algorithms of interest in most parallel applications, their general features and ensuring their deadlock freedom.

The simplest time model of communication in distributed memory systems uses a number of communication steps (rounds): point-to-point communication takes one step between adjacent nodes and a number of steps if the nodes are not directly connected. In the more detailed view, the communication time is composed of a fixed start-up time t_s at the beginning and of a component that is a function of distance d (the number of channels on the route or hops a message has to make), and message length m in certain units (words or bytes). For distance-sensitive store-and-forward (SF) switching we have

$$T_{SF} = t_s + d(t_r + mt_1), \qquad t_r \ll mt_1 \tag{5.1}$$

and for distance-insensitive wormhole (WH) switching

$$t_{WH} = t_s + dt_r + mt_1 \tag{5.2}$$

where t_r is a routing delay plus the switching and inter-router latency and t_1 is per unit-message transfer time. In this linear model of communication we assume no contention for channels.

Further, we have to distinguish between unidirectional (simplex) channels and bi-directional (half-duplex, full-duplex) channels. The number of bi-directional channels between the CPU and a router (ports) that can be engaged in communication simultaneously (1-port or all-port models) has also an impact on number of communication steps and communication times, as well as if nodes can combine/extract partial messages with negligible overhead (combining nodes) or can only re-transmit/consume original messages (non-combining nodes). Finally we have to take into account a switching technique (store-and-forward or wormhole) and a network topology.

A few comments are appropriate on communication among various nodes from the simplest to the most complex type. A single neighbor-to-neighbor communication, multiple neighbor-to-neighbor communications, and a single point-to-point communication are always deadlock free. For multiple simultaneous point-to-point communications (including permutation) the deadlock freedom is ensured by the acyclic property of a channel dependency graph. In that graph nodes correspond to channels and a directed edge connects two nodes if and only if a message coming through the first channel is routed to the second one.

A deadlock-free routing table for 8-processor AMP network from Figure 5.1 and the related channel dependency graph are given in Table 5.1 and Figure 5.2. Channels in the graph are denoted by a source node number and a direction (West, North, South, East). Had we routed a message from node 2 to node 7 via node 6 (instead of node 3), as shown by the dotted path, we would have got a cycle in the graph and a possibility of a deadlock.

In many parallel algorithms we often find certain communication patterns, which are regular in time, in space, or in both time and space; by space we understand spatial distribution of processes on processors. Communications taking place within

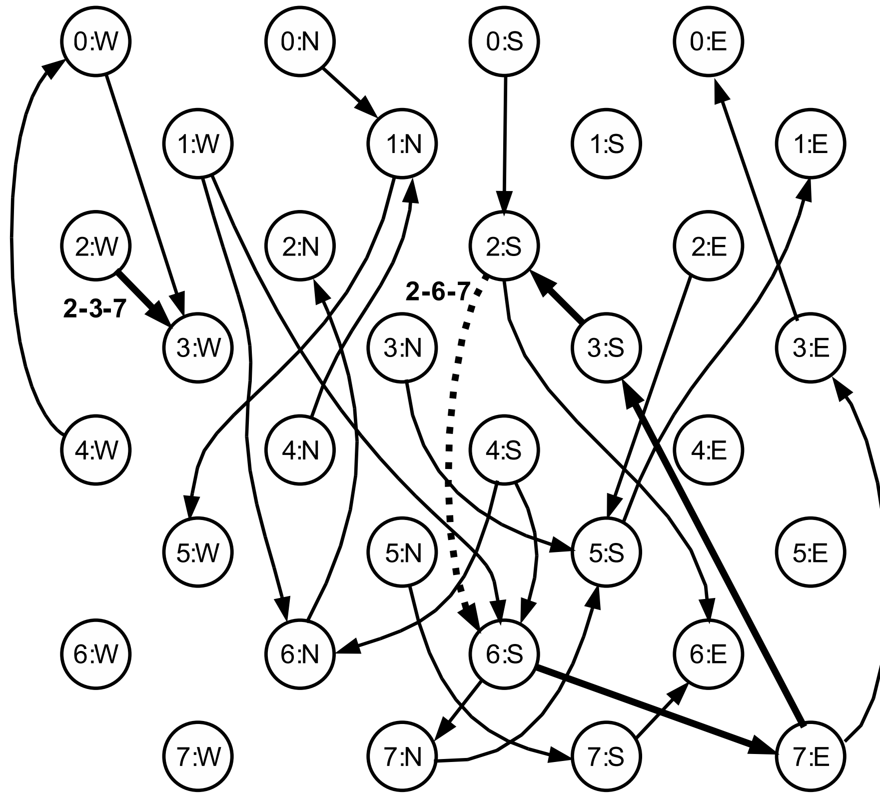


Fig. 5.2: The channel dependency graph

Destination

	PE	0	1	2	3	4	5	6	7	SC
	0	x	N	S	W	E	N	S	W	E
S	1	S	x	W	N	E	N	W	W	E
o	2	N	E	x	W	S	E	S	W	S
u	3	E	N	S	x	E	N	S	W	W
r	4	W	N	S	W	x	N	S	S	E
c	5	W	S	E	W	S	x	N	N	N
e	6	N	W	N	S	E	S	x	S	E
	7	E	N	E	E	S	N	S	x	W
	SC	W	W	W	S	W	S	S	S	x

Tab. 5.1: 8-processor AMP routing table

a subset or the set of all processors are called group or collective communications. Examples of these may serve:

- OAB (One-to-All Broadcast): One node sends the same message to all other nodes.
- OAS (One-to-All Scatter): One node sends distinct messages to all other nodes.

- AOG (All-to-One Gather): A reverse operation to OAS — all nodes send a message to a single node.
- AAB (All-to-All Broadcast): All nodes perform OAB at the same time, each node sends its message to all other nodes.
- AAS (All-to-All Scatter): All nodes perform OAS at the same time, each node sends distinct (private) messages to all other nodes.
- Permutation.

Implementation of group communications is inherently prone to a deadlock. The acyclic property of the channel dependency graph is not sufficient, it applies only to the situation when a group of nodes are sending messages to distinct partners (permutation routing or a subset of it). If we let each node send or receive messages to or from more than one partner in a loop asynchronously, we still face a danger of a so called *fetch deadlock* (at least two nodes execute pending send operation and cannot receive). This is why we use a synchronized communication model and assume that the communication proceeds in synchronized rounds (steps). In store-and-forward networks one round is a set of parallel (simultaneous) hops of packets between adjacent nodes. In WH networks, a round is a set of simultaneously communicating pairs along conflict-free paths; one round takes time given by the slowest communicating pair.

In the rest of this chapter, we will focus especially on OAS and AAS operations on interconnection networks with the following parameters:

- full-duplex links — messages can be transferred in both directions at the same time
- store-and-forward switching — whole packets are buffered in switches
- non-combining nodes — every received packet is sent on separately
- all-port communication facility — all ports of a single node can be used for communication simultaneously.

These communication tasks (OAS, AAS) cause the highest communication traffic and their timing overhead greatly depends on capabilities of particular communication hardware. One possibility how to implement AAS is to use permutations separated by barriers. The number of steps is then $P-1$, where P is processor count. However, this number of steps is too large and can be reduced significantly. Since we deal with irregular topologies, the only way to find optimal or sub-optimal schedules of communication steps is by combinatorial optimization. Next we describe our method of doing it.

5.2 GAroute Algorithm Description

The purpose of *GAroute* as described in [71, 72, 73, 74] is to find a (sub-)optimal schedule of neighbor-to-neighbor communications implementing a given group communication (OAS or AAS), especially for irregular networks, where a schedule cannot

node id	neighbor 1 (north)	neighbor 2 (south)	neighbor 3 (west)	neighbor 4 (east)
0	1	2	3	4
1	5	0	6	4
2	0	6	3	5
3	5	2	7	0
4	1	6	0	SC
5	7	1	3	2
6	2	7	1	4
7	5	6	SC	3

Tab. 5.2: Description of 8-processor AMP topology

be constructed analytically. GAroute performs its computation in two main phases. In the first phase, it searches for all shortest paths among nodes in a particular topology. Inclusion of longer paths as well is controlled by a settable parameter. In the second phase, a genetic algorithm is used to build a schedule from the paths found in the first phase.

5.2.1 Input

At the beginning, the program reads its input data from a specified configuration file. It contains values of the following parameters:

- name of the file with a description of the network topology (its graph)
- type of a communication operation (currently OAS or AAS)
- source node (used only for OAS)
- path length increment
- the target number of communication steps for the communication task
- population size
- number of offsprings to be produced in every generation
- mutation probability
- size of a tournament group

Description of the network topology for which a particular group communication is being optimized is specified in a separate file. This file contains a list of each node's neighbors, where two nodes are considered to be neighbors only if they are connected by a single direct link. Table 5.2 shows sample data which describe eight-processor AMP topology in Figure 5.1. Further description of the other parameters is given in the following sections.

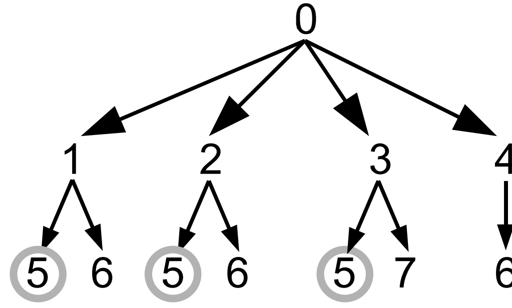


Fig. 5.3: Construction of the shortest paths list from node 0 to node 5 in the 8-processor AMP topology

5.2.2 The Search for the Shortest Paths

The first phase of computation consists of a search for all shortest paths between pairs of nodes. This task is performed by a breadth-first search algorithm: a tree is gradually constructed, one level at a time, from a root that is assigned an index of a source node. When a new level of the tree is generated, every node at the lowest level (leaf) is expanded. When a node is expanded, its successors are determined as all its direct neighbors except those which are already located at higher levels of the tree (this is necessary to avoid cycles). Construction of the tree is finished when a value of at least one leaf is equal to an index of a destination node. Destination leaves' indices determine identity of found paths, which are then stored as sequences of node indices.

A sample tree constructed while searching for shortest paths from node 0 to node 5 in the 8-processor AMP topology is shown in figure 5.3. The *SC* node is not considered. Three paths were actually found in the tree: 0-1-5, 0-2-5 and 0-3-5.

If OAS communication is being scheduled, only paths from a single source node to all other nodes are searched for. On the other hand, for optimization of AAS communication, all paths between every pair of source-destination nodes are considered. Because all links in a target topology are assumed to be bi-directional, to save work only paths to destinations with indices greater than the source node index are sought. All the other paths are constructed by reversing direction of already found paths, i.e. swapping source and destination nodes. All paths are stored in an array and shared by all individuals in a population.

In certain cases when the target topology has nonuniform numbers of links per node, it may happen that an optimal routing schedule cannot be constructed from a set of only the shortest paths. Use of just the shortest paths may cause heavy use of some links while rare use of others, which prevents construction of an optimal solution. To avoid this problem a special parameter *path length increment* has been introduced. This optional parameter specifies a number of hops that can be added to a length of the shortest path. The algorithm then considers not only the shortest paths but also paths whose length may be longer at most by the increment.

5.2.3 The Optimization Algorithm

As soon as the list of the shortest paths (possibly combined with longer ones) is created, it is used for iterative construction of a time schedule of a given group communication by employing a genetic algorithm. An individual's chromosome is represented by an array of genes. Every gene encodes a time schedule of a single message transmission from a given source node to a destination node. This schedule consists of an identity of a path which is used for routing the message and a list of time steps at which every node on the path (except the destination node) sends the message to the next node on the path. Assignment of time steps is constrained by a rule that transmission of any message must not take more steps than a maximum, which is set as an input parameter when the program is run. Number of genes G in every chromosome is determined by the type of communication to be scheduled and by the number of nodes P in a target architecture as

$$G = P - 1 \tag{5.3}$$

for OAS or

$$G = P(P - 1) \tag{5.4}$$

for AAS communication.

At first, an initial population of randomly generated individuals is set up. Then genetic operators (crossover and mutation) are repeatedly applied to individuals in the population to produce offsprings, which replace less fit individuals in the current population. A number of offsprings produced in every generation is set as a configuration parameter. If its value equals to a size of the whole population, then all individuals in the population are replaced by newly generated offsprings in every generation. Otherwise, if the value is lower than the population size, then the fittest individuals from a current population survive for the next generation (so called elitist strategy [59]).

A point of crossover is selected randomly at a boundary of any gene. A multi-point crossover is also possible. Partners for mating are chosen by a tournament selection [60], which is performed by selecting the best individual from a group of randomly selected individuals. A size of the group for a tournament selection can optionally be set as a configuration parameter of the algorithm and it influences a selection pressure toward the best individuals in the population (the larger the group size, the greater the pressure). Mutation is performed after crossover by randomly selecting genes and randomly changing their values (index of a path if more than one is available and also appropriate time steps).

Selection of individuals depends on evaluation of a fitness function. In our representation of chromosomes, fitness of an individual is directly derived from its cost, which is determined by a number of conflicts among genes of its chromosome. A *conflict* is defined as follows: two genes cause a conflict when they encode paths which use the same communication channel in the same time step. It is obvious that we are concerned only with such final solutions which have no conflicts, i.e. whose cost is equal to zero.

This approach to optimization is a bit different from a common use of genetic algorithms when the algorithm searches for (sub-)optimal solutions using an absolute

criterion of optimality. In our case, a user first selects a quality of a sought solution by setting the target number of communication steps S. This parameter is not optimized by GAroute and it is constant during a single execution of the algorithm. A typical work with GAroute then consists of the following steps:

1. Estimate a number of communication steps S required to perform a given group communication.
2. Run the algorithm with this value set as a parameter.
3. If an optimal solution (with cost equal to 0) is found, try to run the algorithm with a lower value of S. Otherwise, if a solution for a given value of S cannot be found in a reasonable time, try to run the algorithm with an increased value of S and possibly tune other parameters as well.

5.2.4 Sequential Performance Tuning

At first, a sequential version of GAroute was implemented in C++ to test ability of the algorithm to provide useful results. To ensure good portability, the implementation uses only standard C and C++ libraries.

Since the first step in achieving a good parallel program performance is single-processor tuning, performance of the sequential program was analyzed to identify bottlenecks. It also appeared that analysis was necessary because test executions of the initial implementation showed that its performance dramatically reduces with increasing number of processors P of the optimized topology. The problem was severe especially for AAS communication, which requires very long chromosomes (the length is given by equation 5.4).

Analysis using profiling tools revealed that a majority of processing time was spent in evaluation of a fitness (objective) function, which determines cost of each individual in every generation. On the contrary, a time taken by genetic operators is negligible. Even though the search for shortest paths (the first phase of the algorithm) requires also significant time, it is performed only once at the beginning of computation and does not affect the overall run-time as much as the genetic algorithm.

The reason of the bad performance was that the initial implementation of the fitness function used a quite trivial approach to determining a cost of a chromosome. Since the cost is defined as a number of conflicts (the same channel used by different genes in the same step) the fitness function counted the conflicts by checking all possible pairs of genes whether they use the same channel for communication. This is repeated for every communication step. So the main part of the function uses two nested loops both iterating through all genes of a chromosome yielding an asymptotic complexity $O(G^2)$, where G is a number of genes determined by the type of communication to be scheduled (equations 5.3 and 5.4 on page 56).

Although some optimizations were already used to simplify a body of each loop, e.g. a set of all genes which already caused a conflict was constructed on the fly so that a single check of the set can avoid expensive path reconstructions and pointer dereferences, the complexity was still unacceptable. Especially if we consider the

AAS communication, which requires long chromosomes (number of genes proportional to P^2), we get a complexity $O(P^4)$ which is infeasible for optimization of architectures with more than a few processors.

To decrease the high complexity, a smarter implementation was developed. Instead of a set of genes which cause a conflict in a given communication step, it uses a set of channels already used in the step. For better performance, every channel is encoded as a single 32-bit integer, which contains indices of the two nodes connected by the channel (higher two bytes for the source node and lower two bytes for the destination node). Use of this data structure simplifies a computation of cost to a single loop iterating through all genes. In every iteration a gene is checked if it uses a channel for communication in the current step. If it does, then the set of used channels is checked for the channel. If the channel is not in the set yet, it is inserted into the set, otherwise a cost in incremented.

Performance of the new implementation is much better since it reduces an asymptotic complexity to $O(G)$ from original $O(G^2)$. Complexity of the loop's body is similar or even smaller than in the original implementation so that a multiplicative constant hidden in the asymptotic complexity formula is decreased as well.

5.2.5 Parallel Performance Tuning

The sequential implementation was successfully applied to smaller scale problems and high quality solutions were obtained. However, optimization of architectures with more than few tens of processors was difficult since large chromosome length (especially for AAS communication) forced use of a smaller population size due to limitations of both a memory size and a processing power, which in turn caused search of a smaller state space and limited quality of found solutions.

To overcome the mentioned limitations, a parallel version was designed, simulated and implemented. The design of an efficient parallel version was supported by data obtained from simulations and from executions of the sequential version. A primary architecture intended for testing the parallel program was a cluster of workstations because it was readily available. Therefore high communication delays were considered during the design. Nevertheless, since the message passing was programmed using MPI library [14] routines, the program can easily be compiled and run on any architecture (clusters of workstations, MPPs, SMPs, etc.) for which an implementation of MPI standard is available.

The Search for the Shortest Paths

It was decided not to parallelize the first phase of the algorithm at all. There were several reasons for this decision:

- Computation of the search for shortest paths takes a small fraction of the overall runtime of the program. Measurements of the sequential code indicate that for populations of at least several tens of individuals, which are quite common, the search lasts approximately as long as a single generation of the genetic algorithm.

- Load balance would be difficult since distances of processors in the optimized topology are not known before the search and therefore trees of different depths would be traversed by different processors.
- Partial results would have to be collected and broadcasted to all processors (AOG + OAB or a single AAB communication), which would cause additional significant overhead.

Due to these difficulties the whole search is performed by all processors. Supposing that they have a local copy of the input file, communication is totally avoided in this phase. Such an implementation assumes that processors with the same performance are used. In a case of heterogeneous parallel computers a more efficient approach would be to let the fastest processor perform the search and then broadcast results to others.

The Optimization Algorithm

Since genetic algorithm (GA) is highly parallel there are several possible approaches to parallelize this problem. Three main categories of parallel genetic algorithms are:

- **Global GA** treats the entire population as a single breeding unit and aims to exploit the algorithmic parallelism inherent in the genetic algorithm.
- **Migration GAs** divide the population into a number of subpopulations, each of which is treated as a separate breeding unit under the control of a conventional GA.
- **Diffusion GA** treats each individual as a separate breeding unit. The second partner for mating is always selected from within a small local neighborhood of the first partner.

Both global and diffusion genetic algorithms were refused because their fine granularity makes them unsuitable for architectures with large communication overhead, which were primarily targeted. The migration GA offers coarse granularity parallelism at the expense that subpopulations are quite isolated and may tend to converge to local optima rather than to global optima. To encourage the proliferation of good genetic material throughout the whole population, individuals migrate between the subpopulations from time to time.

The selected migration approach still leaves some choices to be made, e.g. how often and how many individuals should migrate and if they should be broadcasted to all processors or only to a subset of them (near neighbors). To support qualified choices, a simulation model was developed to ease testing various configurations of the algorithm. It is based on the cluster model described in chapter 4. Some results can be seen in Figure 5.4. The graph shows efficiency of three variants of communication on a cluster of four to sixteen processors. The model simulates migration of ten individuals among subpopulations in every tenth generation of a total of 200 generations.

The version denoted *P-1 neighbors* in the graph simulates broadcast of the best individuals to all processors. It achieves poor scalability due to communication

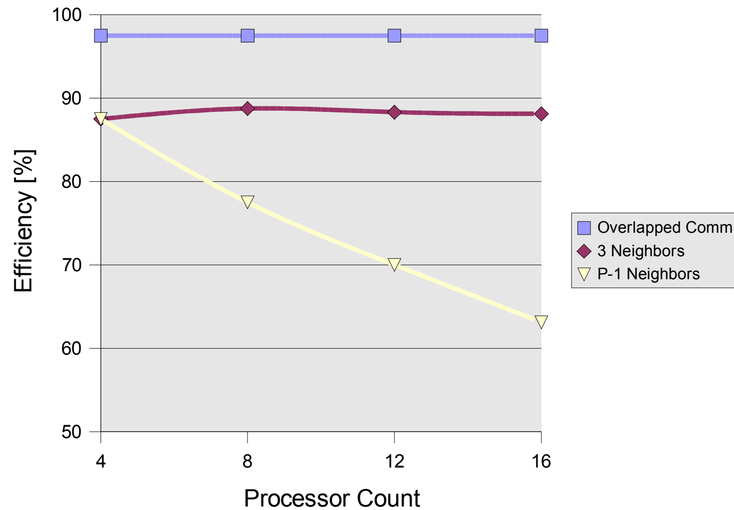


Fig. 5.4: Simulated efficiency of GAroute algorithm for various communication patterns

traffic increasing linearly with the number of processors P while a workload of every CPU remains constant. Such an implementation is therefore unsuitable for larger numbers of processors.

Much better performing is a variant denoted *3 neighbors*, which attains almost linear speedup (constant efficiency). It should be noted, though, that the simulated speedup does not automatically mean similar increase in speed of the algorithm's convergence. It just deals with the communication versus useful computation ratio.

The last case denoted *Overlapped Comm.* simulates efficiency of the algorithm when communication overhead is fully overlapped with useful computation. As expected, the efficiency reaches almost 100 % since the main source of overhead which still remains is the sequential computation performed at the beginning of the algorithm (the search for shortest paths). The dependency will be the sasme for both constant number of neighbors $N = P - 1$ assuming that computation of the generations between two migrations does not take shorter time than communication.

Since in practice 100 % of communication can hardly be performed in background leaving the main processing unit free for other computation and the ratio greatly depends on capabilities of a praticular hardware used and on software implementation, the final solution was to keep $N \ll P$, especially for greater values of P, but still proportional to P. This greatly reduces the communication overhead. In a case of need, the user can still tune the value. Since the cluster used for testing has a star topology, which does not distinguish distances of processors, neighbors were simply determined from indices of the machines.

Despite the mentioned difficulties with overlapping communication, every attempt was made to overlap as much communication with computation as possible. All communications are programmed using MPI nonblocking asynchronous routines. Completion of a previous communication task is checked and possibly waited for only at the point of the next communication so that the previous communication has enough time to complete without waiting.

5.2.6 Experimental Results

The developed program was tested with various parameters (population size, number of offsprings created in every generation, probability of mutation, single-point and double-point crossover). The following paragraphs summarize experience gained during experimentation with these parameters.

It seems that there is no apparent difference in speed of convergence if a double-point crossover is used instead of the simpler single-point crossover. On the other hand, mutation probability has a great impact on convergence. It appears that higher probability (up to 1 percent) causes faster convergence during the first generations, but later, after a few tens or hundreds of generations, frequent mutation slows down convergence significantly.

The best results have therefore been obtained with mutation probability being lowered gradually during computation (a similar process to cooling schedule of Simulated Annealing algorithm [61]). The schedule is based on a minimal cost (cost of the fittest individual in a current population). After minimal cost goes down to a half of its value in the first generation, mutation probability is also lowered to a half. This process is repeated a few times so that last generations are performed with a probability of mutation lowered to a small fraction of its original value at the beginning of computation. Some improvement has also been achieved by mutating more likely those genes which cause conflicts (increasing the cost of their chromosome) than those which do not cause any conflicts.

Population size of one hundred to a few hundred individuals seems to be sufficient. Better results have been achieved when fewer offsprings than the size of the whole population are produced in every generation, so that the fittest individuals from the current population can survive without a danger of being replaced by less fit offsprings. It is usually sufficient to keep two or four fittest individuals.

Scheduling of OAS and AAS communication operations for two different architectures, a hypercube and AMP, was tested. The number of processors in the target architecture varied from 8 to 64. A hypercube has been chosen as a convenient benchmark because of its regular topology with known optimal scheduling. The optimal number of steps S for OAS communication on a hypercube with dimension n and P processors is given by:

$$S = \left\lceil \frac{2^n - 1}{n} \right\rceil = \left\lceil \frac{P-1}{\log P} \right\rceil \tag{5.5}$$

and for AAS communication

$$S = 2^{n-1} = \frac{P}{2} \tag{5.6}$$

OAS and AAS communication complexities, measured by the number of time steps in schedules found by GAroute so far, are shown in Tables 5.3 and 5.4 (columns three and four). The first column gives the node count in the target architectures and the second column shows the number of steps in the known optimal schedule for a hypercube (the reachable lower bound).

The presented data deserve some comments. Firstly, OAS is a quite simple operation and therefore the algorithm is likely to find an optimal solution even for

# of nodes	hypercube optimal	hypercube	AMP
8	3	3	2
16	4	4	–
23	–	–	6
32	7	7	8
42	–	–	11
53	–	–	13
64	11	11	–

Tab. 5.3: Results of OAS optimization

# of nodes	hypercube optimal	hypercube	AMP
8	4	4	4
16	8	9	–
23	–	–	14
32	16	18	22
42	–	–	31
53	–	–	45
64	32	40	–

Tab. 5.4: Results of AAS optimization

larger architectures in a couple of tens of iterations if its parameters are set correctly (mutation probability, population size, ...).

Although AAS has a higher order of complexity than OAS, GAroute has been successfully applied even to this communication pattern. Nearly optimal solutions have already been found for architectures with up to 32 processors and acceptable results have been attained for larger networks. A further improvement of these results can be expected in the future, because a number of experiments, which could be carried out so far, was limited by the overall run time required for optimization (many hours if nearly optimal solutions are sought). On the other hand, if we need an acceptable solution quickly, GAroute can be run with a larger target number of communication steps S and it is likely to find such solution quickly.

To illustrate the results found by GAroute, Table 5.5 contains eight AAS routing tables for eight nodes in AMP topology. Rows show message movement in at most four steps from source nodes (left columns) to destination nodes (bold digits), possibly via intermediate nodes. The asterisk means that the message does not move in the given step. Two intermediate nodes in some rows indicate non-minimum routing over three channels, since the diameter $D = 2$.

Each node stores its table and communicates with its neighbors in four steps (rounds) according to the table. For example node 2 sends messages to neighbors 0 and 3 in the first step, to neighbor 5 in the second step and to neighbor 6 in the third step. Messages to remaining nodes 1, 4, and 7 go via intermediate nodes 5, 6, and 6 and reach destinations in step 4, 2, and 3, respectively. On the other hand, node 2 receives messages from nodes 0 and 3 in step 2, from node 7 in step 3 and from nodes 4, 6, 1, and 5 in the last step 4.

0	*	*	**1**	*
0	*	**2**	*	*
0	**3**	*	*	*
0	**4**	*	*	*
0	*	*	3	**5**
0	*	*	2	**6**
0	*	3	*	**7**

1	4	*	**0**	*
1	5	*	*	**2**
1	0	*	*	**3**
1	*	*	**4**	*
1	*	**5**	*	*
1	*	*	**6**	*
1	6	**7**	*	*

2	**0**	*	*	*
2	*	*	5	**1**
2	**3**	*	*	*
2	6	**4**	*	*
2	*	**5**	*	*
2	*	*	**6**	*
2	*	6	**7**	*

3	*	*	**0**	*
3	0	*	*	**1**
3	*	**2**	*	*
3	*	0	*	**4**
3	*	*	**5**	*
3	7	**6**	*	*
3	5	*	*	**7**

4	*	*	*	**0**
4	*	6	*	**1**
4	*	1	0	**2**
4	6	*	2	**3**
4	*	*	1	**5**
4	*	*	*	**6**
4	*	*	6	**7**

5	*	*	1	**0**
5	**1**	*	*	*
5	*	3	*	**2**
5	*	*	*	**3**
5	2	0	**4**	*
5	*	1	*	**6**
5	*	**7**	*	*

6	4	**0**	*	*
6	*	*	4	**1**
6	*	*	*	**2**
6	*	2	**3**	*
6	1	**4**	*	*
6	*	1	**5**	*
6	**7**	*	*	*

7	*	*	3	**0**
7	6	*	**1**	*
7	*	3	**2**	*
7	*	*	*	**3**
7	*	*	6	**4**
7	*	*	*	**5**
7	*	*	*	**6**

Tab. 5.5: 8-processor AMP routing table

5.3 Conclusions

This chapter presented a design, implementation and performance tuning of a parallel iterative optimization algorithm. The design was supported by the same simulation approach described in previous chapters. A model of a workstation cluster was used to provide a preliminary view of the program's performance on the particular architecture. It enabled quick decisions about various variants of an implementation without a need to implement several different versions only for testing and comparison.

Beside the fact that the developed program serves as a representative case study of the performance tuning approach, the results also indicate usefulness of the designed algorithm in other areas of parallel computing. The algorithm provided high quality schedules of group communications, which cannot be constructed by any heuristic approach known to the author. Support for other types of communication and hardware capabilities of target architectures can be incorporated into the program to further broaden its application area.

Moreover, a user-selectable optimality of sought solutions is an unusual approach not mentioned in literature concerning genetic algorithms. This approach was required to maintain constant length of chromosomes and was successfully employed in the algorithm.

Aside from GAroute, simulations of other optimization algorithms were also performed. For example a diffusion genetic algorithm used for optimization of stack filters in image processing was simulated on a 2D-torus architecture. Achieved results were published in [70].

Chapter 6

Conclusions and Future Research Directions

6.1 Contributions

A primary goal of the thesis was development of a unified approach to modeling of parallel architectures and algorithms with special emphasis on estimation of obtainable performance. The goal has been fulfilled by applying a CSP-based simulation approach, which was originally designed by its authors for prototyping of parallel algorithms on a particular message passing architecture. The modeling approach has been extended to support simulation of a wide variety of architectures and programming paradigms. Advantages of the developed approach include following:

- It can be used to run typical programs on theoretical machines like PRAM and asynchronous PRAM (APRAM) and derive their execution times.
- It can simulate execution of message-passing as well as shared-variable programs on various architectures.
- It can describe interconnection of processors via direct (hypercube, 2D-torus, full connection) as well as indirect networks (multistage interconnection networks), crossbars and buses.
- It can include time model of message passing communication as well as shared-memory communication (possibly the write miss in a sender cache, synchronization, and the read miss in a receiver cache).
- It allows description of various synchronization operations (locks, barriers) and their timing overhead.
- It is able to evaluate performance of parallel applications simultaneously with (simulated) execution.

As results presented in the thesis indicate, a quite satisfactory accuracy of simulation can usually be attained. Especially when measurements of implemented fragments of sequential code were performed and models augmented with the measured data, high accuracy (within 10 %) has been achieved. The accuracy has been

verified for models of two most common parallel architectures — a bus-based symmetrical multiprocessor and a switch-connected cluster, since the two architectures were available for experimentation (a 4-processor SMP and a cluster of workstations).

The approach has also been successfully used in teaching of parallel architectures and algorithms. Students can experiment with the models to better understand fundamental principles of parallel architectures and to get the first experience with debugging of parallel programs. They can also use the described models of various architectures, synchronization operations, etc. to build more complex models and experiment with them. A broad range of parallel algorithms have already been simulated in various student assignments. Main application areas of models available for experimentation include:

- **Sorting and searching** — bitonic sort, parallel sort with regular sampling, merge-sort, parallel search
- **Linear algebra** — matrix transposition and multiplication, Gauss-Jordan method, multidiagonal sparse linear equations, iterative solution of linear equations, FFT, Divide and Conquer paradigm applied to FFT
- **Synchronization** — butterfly and dissemination barriers, LL-SC lock using coherency, a centralized barrier with a shared counter, dining philosophers
- **Soft computing** — multi-layer artificial neural networks, genetic algorithms, cellular neural networks
- **Other** — parallel reduction, scan and broadcast, tree farm, linear farm, prime number generation, deadlock at routing, minimum spanning tree, A Minimum Path (AMP) topology, fat tree interconnection, omega interconnection network

Aside from fulfillment of main goals of the thesis, tuning the performance and optimization of collective communication algorithms for irregular topologies have been demonstrated using a parallel program GAroute. Results presented in section 5.2.6 prove its ability to provide high quality results, which are useful for design of efficient communication algorithms. This parallel program has been itself tuned for the best performance and serves as an example of useful parallelization techniques.

Other particular contributions of this work include following:

1. A general library of basic models (interconnection networks, synchronization mechanisms) has been developed, which can be used for building complex models of parallel systems and applications.
2. The thesis demonstrated general application of `ALT` statement for simulation of various types of arbitration. A central arbiter has been designed, which is fair for arbitrary number of requesters. Moreover, distributed as well as centralized "on demand" arbitration has been implemented.
3. Use of `ALT` statement in models of bidirectional interconnection networks based on synchronous channel communication has been found to be prone to a fetch deadlock. The fact has not been mentioned in other literature concerning Occam or Transim languages yet.

4. A novel variant of Genetic algorithm has been introduced. It uses a progressively adjusted quality metrics of a searched solution and reduces the cost function of individuals not to a minimum but to zero.

6.2 Future Research Directions

Although Transim proved to be a very useful tool for simulations performed in this work, its limitations may make it unsuitable under certain circumstances. Problems appeared especially during simulations of complex models like a cluster of SMPs, which employs a combination of both message passing and shared variable paradigms. Sometimes the limitations could be overcome e.g. by using distributed simulation as mentioned in section 4.4, but in some cases they may be more difficult to handle.

Due to the limitations, possibilities of use of the developed simulation approach in other environments have also been investigated. A promising framework seems to be *JavaCSP* [62], which is a Java implementation of various CSP constructs. It is a class library that allows adoption of CSP approach to concurrency with all its advantages. The models developed in this thesis could therefore be ported to JavaCSP to allow e.g. larger scale simulations than Transim would support.

Another promising direction of further research seems to be performance modeling of *grid computing*. This recent technology tries to provide an uniform way of access to computing resources spread in geographically large areas. It should enable accumulating idle processing power available throughout the Internet, which could then be used for computationally intensive tasks. Attaining efficient computation in such an environment is apparently a difficult task and should be solved with a help of sound modeling and performance prediction tools. An attempt to predict performance of computing in a wide area network environment by use of the described modeling approach has already been published in [66].

Even though the developed and implemented algorithm GAroute is already able to provide useful results, which indicate that main ideas behind the algorithm are correct, a range of its possible applications is still limited. Further development of the program can improve its usability by e.g. providing support for other types of group communications, switching techniques, packet manipulation etc.

Appendix A

Transim Language Reference

This is a short reference of the most frequently used Transim commands. It does not attempt to be a copy of a formal specification of the language but rather to be a quick guide, which makes reading segments of source code of simulation models in the text easier. The reference is divided into three sections: Software Description, Hardware Description and Mapping Software onto Hardware similarly to structuring of source code of the models.

A.1 Software Description

A.1.1 Data Types

The following three primitive data types are available. No real numbers or operations on them are supported.

INT an integer type, its limits depend on a host operating system.

BOOL a boolean type with possible values `TRUE` and `FALSE`.

INTC a constant value which can be shared in hardware and software description.

A (multidimensional) array can be constructed from a primitive type, for example: `[10][5] INT matrix`.

A.1.2 Code Structuring

Blocks of code are designated either by indentation (two space characters) or by braces (‘`{`’ and ‘`}`’). The following statements determine how blocks are executed.

SEQ determines a block of statements which are executed sequentially. Loops can be expressed by replication of `SEQ` (e.g. `SEQ i = 0 FOR n`).

PAR executes following blocks in parallel. Each block must be preceded by `SEQ | name` statement, where `name` determines a name of the process. All processes are executed concurrently on the same processor and in the same priority.

PRI PAR statement in contrast to `PAR` executes processes with different priorities: the first process (in textual order) is executed in high priority, other processes are executed in low priority. Neither `PAR` nor `PRI PAR` statement can be replicated in Transim.

PLACED PAR executes following processes (named blocks) on different processors in parallel. The statement can be replicated (`PLACED PAR i = 0 FOR n`) to provide a SPMD programming model.

A.1.3 Communication

Communication and synchronization means are provided by channels. Channels in Transim use only a single protocol (`ANY`). A channel `ch` is therefore declared as follows: `CHAN OF ANY ch`. Arrays of channels are also supported (declared as: `[n] CHAN OF ANY chn`) and can mix channels between and within processors (hard and soft channels). Communication is expressed by following statements:

! denotes a send operation, e.g. `ch ! expr | mesg.length`, where `expr` is an integer expression which is evaluated and the result is sent to a destination process. `mesg.length` determines simulated length of the sent message.

? denotes a receive operation, e.g. `ch ? var`. A received integer value is stored in a variable `var`.

ALT is a receive statement which selects from multiple input channels. If more than one channel is ready for receive, the channel used for the next communication operation is determined arbitrarily — in Transim always the first channel in textual order.

PRI ALT statement unlike `ALT` always receives from the first ready channel in textual order. In Transim the functionality is the same as that of simple `ALT`.

Two predefined channels can be used in the simulation:

DEVICE is a channel for communication with an external device. The channel can be used for both input and output. Message length must be specified in both cases.

OUTSTRM is an output channel used only for debugging and tracing. Values sent to this channel are printed into an output file by the simulator. Communication on this channel does not influence the simulated time.

A.1.4 Predefined Functions and Procedures

The only function and procedures which can be used are following:

RAND(low, high) is a function which returns randomly generated value in the interval <low, high>.

SERV(expr) defines the length of simulated useful processor work, `expr` is the number of CPU cycles.

WAIT(expr) indicates that a processor is idle for the specified number of cycles `expr`.

NOTE(str) causes string `str` to be printed to an output file by the simulator. It is useful for timing purposes.

A.2 Hardware Description

Each node (processor) of a simulated architecture is specified by a `NODE` statement, which can have following parameters:

SPD processor speed in MHz

LS link speed in Mbit/s

ICS internal channel speed in Mbyte/s

ECS external channel speed in Mbyte/s

ICD internal channel delay in μs

ECD external channel delay in μs

TSL time-slice period in CPU cycles

EF external memory factor (the number of additional CPU cycles for one external memory cycle)

The `NODE` statement can be replicated (e.g. `NODE i = 0 FOR n`). In a case that links of different speeds are used, a `LINK` statement is also required to describe interconnection of nodes. Otherwise, interconnection is automatically obtained by the simulator from a Transim source file. Example:

```
NODE
  NODE i = 0 FOR P
    NODE cpu : SPD = 500, ECD = 485, ECS = 7.9
```

A.3 Mapping Software onto Hardware

Named components of a (replicated) `PLACED PAR` statement are mapped to processors (specified by the `NODE` statement in a hardware description) by a `MAP` statement. The statement can also be replicated. Example:

```
MAP
  MAP i = 0 FOR P
    MAP cpu[i] : cell[i]
```

Bibliography

[1] Moore, G.: *Moore's Law.*
URL: `http://www.intel.com/research/silicon/mooreslaw.htm`

[2] Gill, S.: Parallel Programming. In: *The Computer Journal*, Vol. 1, pp. 2–10, April 1958.

[3] Holland, J.: A Universal Computer Capable of Executing an Arbitrary Number of Sub-programs Simultaneously. In: *Proc. East Joint Computer Conference*, Vol. 16, pp. 108–113, 1959.

[4] Ackland, B., Anesko, A., Brinthaupt, D., Daupert, S. J., Kalavade, A., Knobloch, J., Micca, E., Moturi, M., Nicol, C. J., O'Neill, J. H., Othmer, J., Sackinger, E., Singh, K. J., Sweet, J., Terman, C. J., Williams J.: A Single-Chip, 1.6 Billion, 16-b MAC/s Multiprocessor DSP. *IEEE Journal of Solid-State Circuits*, Vol. 35, No. 3, pp. 412–424, March 2000.

[5] Flynn, M. J.: Very high-speed computing systems. *Proc. IEEE*, Vol. 12, pp. 1901–1909.

[6] Anderson, T. E., Culler, D. E., Patterson, D.: A Case for NOW (Networks of Workstations). *IEEE Micro*, Vol. 15, No. 1 (Feb.), pp. 54–64.

[7] Pacheco, P.: *Parallel Programming with MPI*, Morgan Kaufmann, San Francisco, California.

[8] Myricom, Inc.: *Myrinet Overview.*
URL: `http://www.myricom.com/myrinet/overview/`

[9] Bal, H. E.: *The Orca Parallel Programming Language.*
URL: `http://www.cs.vu.nl/vakgroepen/cs/orca.html`

[10] Hoare, C. A. R.: *Occam 2 Reference Manual*, Prentice-Hall, 1988, ISBN 0136293123.

[11] HPF Forum: *High Performance Fortran: Language Specification*, 1993.

[12] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: *Parallel Programming in OpenMP*, Morgan Kaufmann, 2000.

[13] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. S.: *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing (Scientific and Engineering Computation)*, MIT Press, 1994, ISBN 0262571080.

[14] Snir, M., Otto, S. W., Huss-Ledermann, S., Walker, D. W., Dongarra, J.: *MPI The Complete Reference*, MIT Press, Cambridge, Massachusetts, 1996.

[15] Amdahl, G.: Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities. *Proc. 1967 AFIPS Conf.*, Vol. 30, p. 483.

[16] Gustafson, J. L.: Reevaluating Amdahl's Law. *Comm. ACM*, Vol. 31, No. 1, pp. 532–533.

[17] *SARA research project.*
URL: `http://www.mrtc.mdh.se/cal/cal_old/Reserachproject/`

[18] *SoCrates research project.* URL: `http://www.mrtc.mdh.se/socrates/`

[19] Roda, J. L., Rodriguez, C., Sande, F., Morales, D. G.: *A new model for the analysis of asynchronous parallel algorithms*, Lecture Notes in Computer Science, 1998.

[20] Wu, X.: *Performance Evaluation, Prediction and Visualization of Parallel Systems*, Kluwer Academic Publications, 1999.

[21] Valiant, L. G.: A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[22] Culler, D., Karp, R., Patterson, D.: LogP: Towards a Realistic Model of Parallel Computation. In: *Proceedings of the 4th ACM SIGPLAN Conference on Parallel Programming Practice and Experience (PPOPP)*, ACM, May 1993, 1–12.

[23] Vrsalovic, D. F., Siewiorek, D. P., Segall, Z. Z., Gehringer, E. F.: Performance Prediction and Calibration for a Class of Multiprocessors, *IEEE Trans. Comput. 37*, 11 (Nov.), pp. 1353–1365, 1988.

[24] Heildelberger, P., Triveli, K. S.: Analytic Queuing Models for Programs with Internal Concurrency. *IEEE Trans. Comput. C-32*, 1 (Jan.), pp. 73–82, 1983.

[25] Ammar, R. A., Qin, B.: A Technique to Derive the Detailed Time Costs of Parallel Computation. In: *Proceedings of the 12th Annual International Computer Software and Application Conference*, pp. 113–119, 1988.

[26] Clement, M. J., Quinn, J.: *Analytic Performance Prediction on Multicomputers*, Technical Report, Department of Computer Science, Oregon State University, Mar. 1993.

[27] Parashar, M., Hariri, S.: Interpretive Performance Prediction for Parallel Application Development. *Journal of Parallel and Distributed Computing*, pp. 17–47, Jan. 2000.

[28] Skadron, K., Martonosi, M., August, D. I, Hill, M. D., Lilja, D. J., Pai, V. S.: Challenges in Computer Architecture Evaluation. In: *Computer*, Vol. 36, pp. 30–36, August 2003.

[29] Culler, D. E., Singh, J. P., Gupta, A.: *Parallel Computer Architecture*, Morgan Kaufmann Publishers, Inc., 1999.

[30] *MulSim simulator*.
URL: `http://heather.cs.ucdavis.edu/~matloff/mulsim.html`

[31] *SystemC*. URL: `http://www.systemc.org/`

[32] Baghdadi, A. et al.: Design Space Exploration for Hardware/Software Co-design of Multiprocessor Systems. In: *11th IEEE International workshop on Rapid System Prototyping*, Paris, France, June 21–23, 2000.

[33] Fahringer, T., Gerndt, M., Riley, G., Traff, J. L.: *Knowledge Specification for Automatic Performance Analysis*, APART Technical Report, `www.fz-juelich.de/apart`, 2001.

[34] Fahringer, T., Gerndt, M., Riley, G., Traff, J. L., Formalizing OpenMP Performance Properties with the APART Specification Language (ASL). *International Workshop on OpenMP: Experiences and Implementation*, Lecture Notes in Computer Science, pp. 428–439, Springer Verlag, Tokyo, Japan, October, 2000.

[35] Fahringer, T., Gerndt, M., Riley, G., Traff, J. L., Specification of Performance Problems in MPI-Programs with ASL. *International Conference on Parallel Processing (ICPP '00)*, pp. 51–58, 2000.

[36] Scherger, M., Baker, J., Potter, J.: Using the UML to Describe the BSP Model of Parallel Computation. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, US, CSREA, 2002, ISBN 1-892512-88-2.

[37] Pllana, S., Fahringer, T.: On Using UML to Model Shared Memory and Message Passing Programs. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, US, CSREA, 2002, ISBN 1-892512-88-2.

[38] Fahringer, T., Pllana, S.: *Performance Prophet*, Institute for Software Science, University of Vienna.
URL: `http://www.par.univie.ac.at/project/prophet`

[39] Hart, E.: *TRANSIM — Prototyping Parallel Algorithms*, 2nd Edition, London, University of Westminster Press, 1999.

[40] Hoare, C. A. R.: *Communicating Sequential Processes*, 2nd Edition, Prentice Hall, 2003. URL: `http://www.usingcsp.com/cspbook.pdf`

[41] Schneider, S.: *Concurrent and Real-time Systems: The CSP Approach*, Wiley, 1999, ISBN 0-471-62373-3.

[42] Roscoe, A. W.: *The Theory and Practice of Concurrency*, Prentice Hall International Series in Computer Science, 1997, ISBN 0-13- 674409-5.

[43] Fortune, S., Wyllie, J.: Parallelism in Random Access Machines. *Proc. 10th Symp. Theory Computing*, ACM, New York, pp. 114–118.

[44] Heywood, T., Leopold, C.: *Models of Parallelism, Abstract Machine Models for Highly Parallel Computers*, Oxford Science Publications, Oxford, England, 1995.

[45] Keller, J., Kessler, C. W., Träff, J. L.: Practical PRAM Programming. *Wiley Interscience Series on Parallel and Distributed Computing*, Wiley, New York, 2001, ISBN 0-471-35351-5.

[46] Knuth, D. E.: Sorting and Searching, volume 3 of *The Art of Computer Programming*, Addison-Wesley, 1973.

[47] Lee, J.-D., Batcher, K. E.: Minimizing Communication in the Bitonic Sort. In: *IEEE Transactions on Parallel And Distributed Systems*, Vol. 11, No. 5, May 2000.

[48] Bracha, G., Gosling, J., Joy, B., Steele, G.: *The Java Language Specification, Second Edition*, Addison Wesley, June 2000, ISBN 0201310082.
URL: `http://java.sun.com/docs/books/jls/`

[49] Andrews, G. R.: *Foundations of Multithreaded, Parallel, and Distributed Programming*, pp. 302, 292, Addison Wesley, 2000.

[50] Cooley, J. W., Tukey, J. W.: An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.*, Vol. 19, pp. 297–301, 1965.

[51] Gupta, A., Kumar, V.: The Scalability of FFT on Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 8, pp. 922–932, 1993.

[52] Norton, A., Silberger, A. J.: Parallelization and Performance Analysis of the Cooley-Tukey FFT Algorithm for Shared-Memory Architectures. *IEEE Transactions on Computers*, Vol. C-36, No. 5, pp. 581–591, 1987.

[53] Zomaya, A.: *Parallel and Distributed Computing Handbook*. pp. 344, McGraw Hill, 1996.

[54] Duato, J., Yalamanchili, S., Ni, L.: *Interconnection Networks*, Morgan Kaufmann publishers, SF, USA, 2003, ISBN 1-55860-852-4.

[55] Fogel, D. B.: *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, IEEE Press, 1995.

[56] Goldberg, D. E., Genetic Algorithms in Search, Optimization and Machine Learning. *Complex Systems*, pp. 153–171, 1989.

[57] Chalmers, A., Tidmus, J.: *Practical Parallel Processing*, International Thomson Computer Press, 1996.

[58] Hwang, K., Xu, Z.: *Scalable Parallel Computing Technology, Architecture, Programming*, McGraw Hill, 1998.

[59] De Jung, K. A.: *An analysis of the behavior of a class of genetic adaptive systems*, Doctoral dissertation, University of Michigan, Univerrsity Microfilms No. 76-9381, 36(10), 1975.

[60] Goldberg, D. E.: A note on Boltzmann tournament selection for genetic algorithms and population-oriented simulated annealing. *Complex Systems*, pp. 445–460, 1990.

[61] Sait, S. M., Youssef, H.: Iterative Computer Algorithms with Applications in Engineering, Solving Combinatorial Optimization Problems. *IEEE Computer Society*, pp.66-73, Los Alamitos, California, USA, 2000.

[62] *JavaCSP project.* URL: `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`

Author's Publications

[63] Dvořák, V., Staroba, J.: Simulating PRAM algorithms with TRANSIM. In: *Proceedings of the 35th Spring International Conference Modelling and Simulation of Systems MOSIS 2001*, pp. 81–86, Hradec nad Moravicí, CZ, MARQ, 2001.

[64] Dvořák, V., Staroba, J.: Numerical Performance Models of Synchronization Operations in Shared-Variable Programs. In: *Proceedings of XXIII Int. Autumn Colloquium ASIS 2001*, pp. 65–70, Ostrava, CZ, MARQ, 2001.

[65] Staroba, J., Dvořák, V.: Simulation-Based Performance Tuning of a Parallel Bitonic Sort Algorithm. In: *Proceedings of the 7th Conference Student FEI 2001*, pp. 358–362, Brno, CZ, VUT, 2001.

[66] Čejka, R., Dvořák, V., Staroba, J.: Predicting performance of SMP clusters. In: *Distributed and Parallel Systems – Cluster and Grid Computing*, pp. 38–45, Boston/London, US, Kluwer, 2002, ISBN 1-4020-7209-0.

[67] Staroba, J, Dvořák, V.: Design Space Exploration of Parallel Embedded Applications Based on Performance-Oriented Specifications. In: *Proceedings of the Joint Workshop on Formal Specifications of Computer-Based Systems*, pp. 71–75, Stirling, GB, US, 2002.

[68] Dvořák, V., Staroba, J.: Performance Prediction Model of Bus-Based Shared Memory Architectures. In: *Proceedings of 36th International Conference MOSIS'02 Modelling and Simulation of Systems*, pp. 273–280, Ostrava, CZ, MARQ, 2002.

[69] Staroba, J., Dvořák, V.: Parallel Linear Equations Solvers for Scientific Simulation: Cluster and SMP Experience. In: *Proceedings of XXIVth International Autumn Colloquium ASIS'02 Advanced Simulation of Systems*, pp. 225-230, Ostrava, CZ, MARQ, 2002, ISBN 80-85988-77-1.

[70] Staroba, J.: Optimization of a Fully Distributed Stack Filter Using Genetic Algorithms. In: *Proc. ot the 8th conference Student EEICT*, pp. 501–505, Brno, CZ, FEKT VUT, 2002.

[71] Dvořák, V., Staroba, J.: Genetic Search for the Shortest Group Communications on Irregular Topologies. In: *Proceedings of XXVth International Autumn Colloquium ASIS 2003*, pp. 321-326, Ostrava, CZ, MARQ, 2003, ISBN 80-85988-88-7.

[72] Staroba, J., Dvořák, V.: Genetic Algorithm Optimization of Group Communications. In: *9th International Conference on Soft Computing Mendel 2003*, pp. 47-52, Brno, CZ, FSI VUT, 2003, ISBN 80-214-2411-7.

[73] Staroba, J.: Collective Communication Scheduling for Parallel Computers. In: *Proceedings of 37th International Conference MOSIS'03 Modelling and Simulation of Systems*, pp. 65-70, Ostrava, CZ, MARQ, 2003, ISBN 80-85988-86-0.

[74] Staroba, J., Dvořák, V.: Design of Low-Cost Communication Algorithms for Irregular Networks. In: *Proceedings of the 3rd International Conference on Networking ICN '04*, pp. 980–985, Guadeloupe, French Caribbean, Feb. 2004, ISBN 0-86341-325-0.

List of Figures

1.1 Basic structure of a shared-memory multiprocessor 3
1.2 Basic structure of a distributed-memory multiprocessor 4
1.3 State diagram of process transitions in Transim 12

2.1 Bitonic sort algorithm 19
2.2 Simulated run time and efficiency of 3 variants of bitonic sort 20
2.3 Comparison of simulated run time and efficiency of sort++ on PRAM and a bus-based SMP 21

3.1 MSI and MESI cache coherence protocols 23
3.2 Clients/SM server model of a cache-coherent, bus-based, SM multiprocessor 24
3.3 Clients and SM server with three operations, software description . . 25
3.4 The essential part of `Test-and-Test&Set` lock model 28
3.5 Array lock — Transim code 29
3.6 Transim code for a sense-reversal barrier with an explicit overhead model 30
3.7 A dissemination barrier — Transim code 31
3.8 A global time [μs] of bus transfers for various locks: 1) Test & Set, 2) Test-and-Test & Set), 3) Array lock, 4) Ticket lock 32
3.9 Barrier-related overhead for a dissemination barrier (D-bar) and a counter-based barrier (C-bar) 33
3.10 Parallel FFT for $N = 16$ and $P = 4$ 34
3.11 The code for simulation of guided loop scheduling 35

4.1 Omega network topology and a process graph of one of its switch modules 38
4.2 Transim code of an input process running in an Omega network's switch module 38
4.3 Transim code of an output process running in an Omega network's switch module 39
4.4 Fat tree topology 40
4.5 Process graph of a crossbar switch model 40
4.6 Model of a cluster of SMP nodes connected by a crossbar switch — a process graph 43
4.7 A matrix of an algebraic system in four phases of Gauss-Jordan elimination algorithm 45

4.8 Four different states of matrix A during computation of Gauss-Jordan elimination method 45
4.9 Speedup of real execution of a linear equation benchmark program and achieved relative accuracy of simulation 46
4.10 Execution times and speedup for simulated architectures with limited L1 cache size 48

5.1 Eight-processor AMP configuration 50
5.2 The channel dependency graph 52
5.3 Construction of the shortest paths list from node 0 to node 5 in the 8-processor AMP topology 55
5.4 Simulated efficiency of GAroute algorithm for various communication patterns 60

List of Tables

3.1 Results of real and simulated 1D N-point parallel FFT on 4 processors 36

4.1 Speedup for simulated (S_{sim}) and real execution (S_{real}) of a linear equation benchmark program on a cluster of workstations 46
4.2 Execution times and speedup for simulated architectures with limited L1 cache size 47

5.1 8-processor AMP routing table 52
5.2 Description of 8-processor AMP topology 54
5.3 Results of OAS optimization 62
5.4 Results of AAS optimization 62
5.5 8-processor AMP routing table 63

Shrnutí

Tato práce se zabývá jednotným přístupem k modelování paralelních architektur a algoritmů s důrazem především na odhad dosažitelné výkonnosti. Pro tento účel byl zvolen modelovací jazyk *Transim*, který byl svými autory původně navržen jako simulátor transputerů vhodný pro prototypování a vyhodnocování výkonnosti paralelních programů komunikujících pomocí zasílání zpráv. V této práci je však použit pro simulaci mnoha různých typů paralelních architektur a programových vzorů, což jde daleko za rámec jeho původně zamýšlených aplikací.

Navržený přístup je předveden na simulacích jak abstraktních modelů, jako jsou PRAM nebo APRAM, tak také běžně používaných paralelních architektur například symetrických multiprocesorů, svazků pracovních stanic a jejich kombinací. Je demonstrováno ladění výkonnosti paralelních algoritmů a výsledky dosažené při simulacích jsou porovnány s výsledky dosaženými na reálných paralelních počítačích. Prezentované simulační modely zahrnují také různé synchronizační operace běžně používané v mnoha paralelních algoritmech, které lze dále použít pro stavbu složitějších modelů.

Bylo provedeno také ladění výkonnosti komunikačních algoritmů, protože tyto algoritmy významně ovlivňují režie paralelních výpočtů. Poněvadž komunikace je nepostradatelnou součástí jakéhokoli paralelního výpočtu, jsou dosažené výsledky použitelné pro širokou oblast paralelních aplikací běžících na distribuovaných strojích využívajících propojovací sítě s nepravidelnou topologií.