
Lappeenrannan teknillinen korkeakoulu
Lappeenranta University of Technology

Pentti Huttunen

DATA-PARALLEL COMPUTATION IN PARALLEL AND DISTRIBUTED ENVIRONMENTS

Acta Universitatis
Lappeenrantaensis 135

ISBN 978-952-214-894-0 (PDF)

Lappeenrannan teknillinen korkeakoulu
Lappeenranta University of Technology

Pentti Huttunen

DATA-PARALLEL COMPUTATION IN PARALLEL AND DISTRIBUTED ENVIRONMENTS

Thesis for the degree of Doctor of Science (Technology) to be presented with due permission for the public examination and criticism in the Auditorium of the Student Union House at Lappeenranta University of Technology, Lappeenranta, Finland on the 4th of December, 2002, at noon.

Acta Universitatis Lappeenrantaensis 135

Supervisor Professor Jari Porras
Lappeenranta University of Technology
Laboratory of Telecommunication
Lappeenranta, Finland

Reviewers Professor Thomas Ludwig
University of Heidelberg
Faculty of Mathematics and Computer Science
Heidelberg, Germany

Kimmo Koski
Nokia Research Center
Helsinki, Finland

Opponents Associate Professor Anne Elster
Norwegian University of Science and Technology
Department of Computer and Information Science
Trondheim, Norway

Kimmo Koski
Nokia Research Center
Helsinki, Finland

ISBN 951-764-700-X
ISSN 1456-4491

Lappeenrannan teknillinen korkeakoulu
Digipaino 2002

Preface

The work for this thesis was done between 1997 and 2002 in the laboratory of telecommunication at the Lappeenranta University of Technology. Financially, this work was supported by the Finnish Cultural Foundation, South Carelian Cultural Foundation, the Research Foundation of the Lappeenranta University of Technology (Lahja and Lauri Hotinen's Fund), the Nokia foundation, and the Lappeenranta University of Technology.

I would like to thank my supervisor, professor Jari Porras, for all his support and help over the past 5 years. Without his enthusiasm and resources I would not have been able to complete my studies and the research required for this thesis. I would also like to express my gratitude to professor Porras for numerous opportunities to attend conferences and to visit the university in Lappeenranta. Also, I would like to thank the two reviewers, Kimmo Koski and Thomas Ludwig, for their valuable comments and suggestions for improving this thesis.

In addition, I would like to acknowledge the help of Jouni Ikonen with respect to my studies, publications, and comments. A special thank you to Jani Peusaari for supporting the workstation cluster in which I conducted most of the experiments for this thesis. Last, but definitely not least, I would like to thank my wife, Annika, for proofreading this thesis and for being supportive through the many evenings and weekends that I spent working on my thesis.

Pentti Hultunen

November 2002
Vancouver, Canada

Abstract

Pentti Huttunen
DATA-PARALLEL COMPUTATION IN PARALLEL AND DISTRIBUTED ENVIRONMENTS
Lappeenranta, 2002
84 p.

Acta Universitatis Lappeenrantaensis 135
Diss. Lappeenranta University of Technology

ISBN 951-764-700-X
ISSN 1456-4491

The past few decades have seen a considerable increase in the number of parallel and distributed systems. With the development of more complex applications, the need for more powerful systems has emerged and various parallel and distributed environments have been designed and implemented. Each of the environments, including hardware and software, has unique strengths and weaknesses. There is no single parallel environment that can be identified as the best environment for all applications with respect to hardware and software properties.

The main goal of this thesis is to provide a novel way of performing data-parallel computation in parallel and distributed environments by utilizing the best characteristics of difference aspects of parallel computing. For the purpose of this thesis, three aspects of parallel computing were identified and studied.

First, three parallel environments (shared memory, distributed memory, and a network of workstations) are evaluated to quantify their suitability for different parallel applications. Due to the parallel and distributed nature of the environments, networks connecting the processors in these environments were investigated with respect to their performance characteristics.

Second, scheduling algorithms are studied in order to make them more efficient and effective. A concept of application-specific information scheduling is introduced. The application-specific information is data about the workload extracted from an application, which is provided to a scheduling algorithm. Three scheduling algorithms are enhanced to utilize the application-specific information to further refine their scheduling properties. A more accurate description of the workload is especially important in cases where the workunits are heterogeneous and the parallel environment is heterogeneous and/or non-dedicated. The

results obtained show that the additional information regarding the workload has a positive impact on the performance of applications.

Third, a programming paradigm for networks of symmetric multiprocessor (SMP) workstations is introduced. The MPIT programming paradigm incorporates the Message Passing Interface (MPI) with threads to provide a methodology to write parallel applications that efficiently utilize the available resources and minimize the overhead. The MPIT allows for communication and computation to overlap by deploying a dedicated thread for communication. Furthermore, the programming paradigm implements an application-specific scheduling algorithm. The scheduling algorithm is executed by the communication thread. Thus, the scheduling does not affect the execution of the parallel application. Performance results achieved from the MPIT show that considerable improvements over conventional MPI applications are achieved.

Keywords: Parallel environments, data-parallelism, scheduling algorithms, MPI, threads, parallel programming paradigms, MPIT.

UDC 004.272 : 004.75

List of Publications

1. Huttunen P., Porras J., and Ikonen J.: Analysis of Parallel Environments for Mobile Network Simulation. In Proceedings of European Simulation Symposium (ESS), Hamburg, Germany, September 28-30, 2000, pp. 164-168.
2. Huttunen P., Ikonen J., and Porras J.: The Impact of Communication in Distributed Simulation. In Proceedings of European Simulation Symposium (ESS), Marseille, France, October 18-20, 2001, pp. 111-115.
3. Huttunen P., Porras J., and Ikonen J.: A Study of Threads and MPI libraries for Implementing Parallel Simulation. In Proceedings of European Simulation Symposium (ESS), Hamburg, Germany, September 28-30, 2000, pp. 96-102.
4. Huttunen P, Ikonen J., and Porras J.: Enhancing Load Balancing in a Data-Parallel GSM Network Simulator through Application-Specific Information. In Proceedings of Conference on Applied Parallel Computing (PARA), Helsinki, Finland, June 15-18, 2002, pp. 542-554.
5. Huttunen P., Ikonen J., and Porras J.: MPIT – Communication/Computation Paradigm for Networks of SMP workstations. In Proceedings of Conference on Applied Parallel Computing (PARA), Helsinki, Finland, June 15-18, 2002, pp. 160-171.
6. Porras J., Huttunen P., Ikonen J.: Accelerating Ray Tracing Based Cellular Radio Coverage Calculation by Parallel Computing Techniques. Annual Review of Communications, Vol. 53, 2000.
7. Huttunen P., Ikonen J., and Porras J.: Parallelization of a WCDMA System Simulator for a Shared Memory Multiprocessor Machine. In Proceedings of European Simulation Symposium (ESS), Erlangen-Nuremberg, Germany, October 26-28, 1999, pp. 556-560.

List of Abbreviations

Abbreviation	Meaning
BSP	Bulk Synchronous Programming
GSM	Global Standard for Mobile communication
MPI	Message Passing Interface
MPIT	MPI-Threads
NOW	Network Of Workstations
PE	Processing Element
PVM	Parallel Virtual Machine
SMP	Symmetric MultiProcessor

List of Terms

Term	Meaning
Completion time	Initialization time of a communication operation. Completion time is measured from the moment a communication (send/receive) function is called until the function returns.
Data-parallelism	Method of parallel computation. All processors execute the same code with either the same or different parameters.
Latency	One-way end-to-end communication time. Latency indicates the total communication time to transfer a message from one processor to another.
Load balancing	Dynamic scheduling. Distribution of workunits occurs during the computation, either when a new workunit is generated, or when a processor requests a workunit.
Partitioning	Division of a workload into workunits. Workunits may be equal or different in size, i.e. have different computational requirements. Furthermore, the workunits may have dependencies between each other requiring processor to interact with each other while processing the workunits.
Scheduling	Distribution of workunits among processors.
Throughput	Bandwidth utilization of a network. Throughput varies based on the protocol and the sizes of the messages used.
Workload	Data that needs to be processed to complete the computation.
Workunit	Piece of workload created in the partitioning phase.

Table of Contents

Chapter 1. Introduction

The last two decades have seen a considerable increase in the demand of computing power. A number of application areas, such as weather prediction, nuclear reaction modeling, and gene mapping, are imposing computational requirements that cannot be fulfilled by single processor systems. Even if uniprocessor systems were capable of executing the applications, their execution times would be excessive. The demand for more computing power is likely to continue over the coming years due to the fact that computers are being used in virtually every field. Three characteristics can be identified that are required from a high performance system to facilitate the execution of complex applications:

1. *Processing power.* Applying more than one processor enables an application to be run simultaneously on a number of processors thereby reducing its execution time.

2. *Memory capacity.* The system has to contain enough memory and disk space to accommodate the data required to store all application data.

3. *Communication network.* In a multiprocessor system, processors are connected to each other, and possibly to a global memory, via an interconnecting network that facilitates data transmissions (message passing and access to remote disks).

Currently, a computer system needs to be equipped with multiple processors in order for it to run complex applications in reasonable time frames. With multiple processors, the previously mentioned requirements can be fulfilled: several processors provide adequate processing capability and the combined memories of the processors are capable of facilitating the application data. In theory, the addition of processors to a system to speed up the execution of an application is quite simple. However, a multiprocessor system introduces a number of issues that need to be addressed in order to efficiently utilize all available processors. Furthermore, since the processors need to exchange data, communication is introduced. The communication plays a significant role in a multiprocessor system, since it can rarely be avoided. The time spent on communication slows down the execution since processors need to partake in the sending and receiving of data.

Parallel execution introduces a number of new software dilemmas that do not exist in sequential applications. The processed data needs to be divided into (preferably independent) parts, and the parts need to be distributed among the processors. The former is accomplished by utilizing a partitioning algorithm, whereas the distribution is handled by a scheduling algorithm. The scheduling algorithm requires communication among processors, as it has to send them the distribution information. The significance of both algorithms cannot be overestimated, since achieving an equal work balance among the processors is important in order to optimize the execution time. In addition, during the execution of the application and at its completion the processors may need to exchange data and collect the sub-results to a

single processor that produces the final results. The two operations introduce further communication.

The above-mentioned problems with parallel applications have varying degrees of difficulty in various parallel environments. It is quite common that a good scheduling algorithm for a specific environment is not the optimal algorithm in another environment. Mainly, the environments determine how the processors are connected to each other and how the data exchange is carried out. For the purpose of this thesis three parallel environments are distinguished: shared memory, distributed memory and a combination of shared and distributed memories (a network of workstations, NOW).

1.1. Background

The research of parallel computing has been conducted at the Lappeenranta University of Technology since the beginning of the 1990's [Por95][Por98a][[Por98b]. The emphasis of this research has been on parallel simulation techniques [Por98c] and communication models [Iko01]. The author's projects have been focused on researching parallelization techniques, parallel programming paradigms, and scheduling in different parallel and distributed environments.

The need for this research emerged when two simulators developed by Nokia Research Center exhibited excessive execution times in a single processor workstation thereby making the required interactive usage of the simulators impossible for their users. The first simulator, known as a GSM network simulator, is a tool enabling network providers to find optimal locations for their base stations in urban environments [Sip96]. The second simulator, a WCDMA system simulator, was designed to be a platform for 3G network studies at Nokia [Kur00].

The author was assigned to the parallelization projects of both simulators in the years 1997 and 1998. The parallel versions of the simulators were implemented based on the SPMD (Single Program Multiple Data) model [Cre02]. Nokia did not impose any hardware requirements for the projects. Therefore, implementations of the GSM network simulator were produced for the three environments (shared memory, distributed memory, network of workstations). For the WCDMA system simulator the only parallel implementation was for the shared memory environment due to the short availability of the system simulator code. In addition to the implementations, further research efforts were carried out to determine optimal scheduling algorithms for the GSM network simulator in all three environments. A concept of application-specific scheduling was developed, which considers information extracted from the application as part of the scheduling procedure. The work in networks of workstations indicated that optimal results could not be achieved with programming paradigm currently available; this prompted the requirement to implement a new programming paradigm for the NOW environment. Therefore, the MPIT paradigm was designed and implemented.

1.2. Objectives

The objective for this thesis was to study methods for increasing the performance of parallel applications by utilizing the best characteristics of three parallel environments as well as the information extracted from applications.

First, the parallel environments were studied and compared to quantify their impacts on parallel computation and programming paradigms. The study showed that no single environment supersedes all other environments for all kinds of parallel applications. The environment has a significant impact on the execution and the performance of an application. Therefore, it was concluded that the selection of an environment is an important issue to consider prior to implementing a parallel application. It was also determined that each environment requires a unique implementation due to differences in characteristics. Second, scheduling algorithms were designed and implemented to take advantage of application-specific information extracted from the applications. In order to improve the capability of a scheduling algorithm it is necessary to retrieve information from the application about the workunits. Third, a programming paradigm for networks of SMP workstations was designed and implemented. The need for such a paradigm became obvious through the study of environments and scheduling algorithms. The programming paradigm, called the MPIT, combines the MPI and the POSIX threads in order to provide an optimal programming environment for a rather complex hardware environment.

The goal of this thesis was further refined by the research projects to reflect the results achieved. Thus, the contribution of this thesis is:

> *To study parallelization methods, scheduling, and communication in data-parallel GSM network and WCDMA system simulators in shared and distributed memory environments, and networks of workstations, and to develop efficient, effective, and, universal scheduling algorithms for all environments, and a programming paradigm for SMP NOWs (the MPIT library).*

1.3. Scope of the Thesis

This thesis discusses the properties of efficient parallel computing in shared and distributed memory environments, and networks of workstations. First, properties imposed by parallel environments are considered. Second, efficient scheduling algorithms are introduced that utilize application-specific information in order to optimize the work balance among the processors. Third, the MPIT programming paradigm is discussed, focusing on writing efficient code in networks of symmetric multiprocessor workstations (SMP NOWs).

Two mobile network simulators were used as case studies for evaluating the properties and suitability of sequential simulators for parallelization. Parameters affecting the parallelization and the performance of the parallelized simulators were studied. The scheduling algorithms were implemented in both the simulators allowing comparisons to be made between the simulators and environments in which they were run.

Figure 1-1 illustrates the different aspects of parallel computation. At the environment level, there are three different parallel environments, distinguished by their memory architectures. The three environments were compared by measuring their communication performance; measurements for completion time, latency and throughput were carried out. The performance results and their impact on parallel computation are evaluated and discussed in [Publication 1][Publication 2].

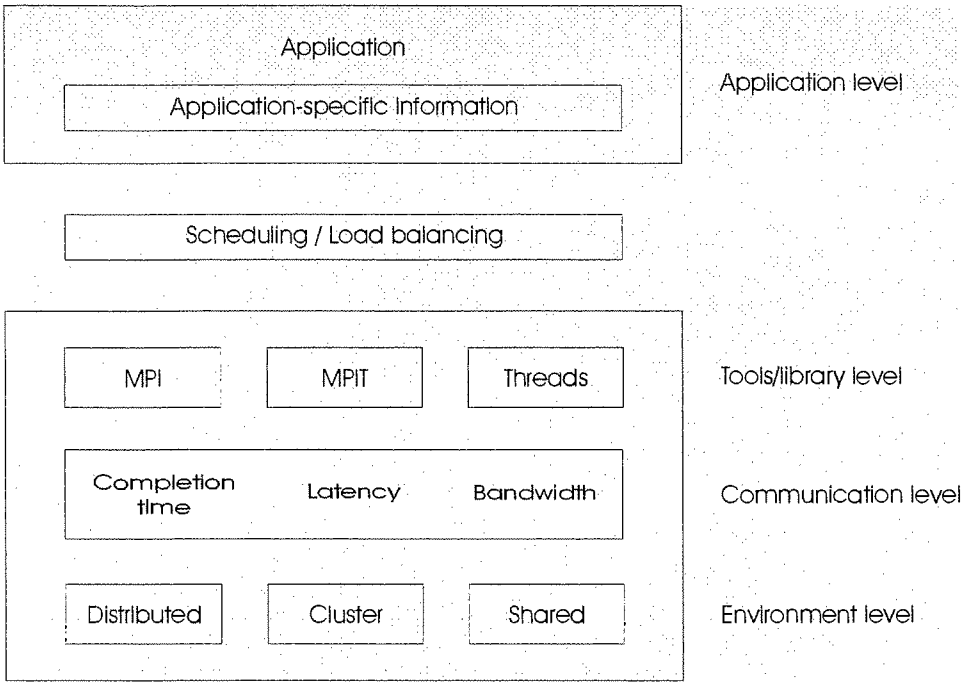


Figure 1-1. The graphical illustration of the problem situation.

At the application level, there is an application that is supposed to be run in parallel. From the software point of view, performance of a parallelized application depends on the structure of the application. The structure defines how the parallelization can be performed, assuming that a sequential version of the application exists. If the structure is not suitable for parallelization, the performance of the application remains low. Furthermore, the application should be able to provide information about the data that is processed in parallel. This information is required by a scheduling algorithm that distributes the data among processors. This thesis introduces a concept called *application-specific scheduling*, which aims to optimize work distribution in parallel environments. Case studies of two mobile network simulators are presented in [Publication 6][Publication 7]. The case studies introduce the simulators and their parallel implementations. [Publication 4] concentrates on scheduling with application-specific information.

At the tools/library level, there are programming paradigms for different parallel environments. The MPI is a message passing interface that allows processes within and among workstations to communicate with each other [MPI95][MPI97][Gro99]. The MPI is used in distributed memory and cluster environments. On the other hand, threads are entities that run user code within a process [But97][Nor96]. Threads are suitable for shared memory environment where the thread communication is performed via a global memory. [Publication 3] examines the suitability of the MPI and the POSIX threads for parallel computing in different environments. A network of workstations is a combination of shared and distributed memory environment especially if there are SMP workstations in the network. The MPI is a suitable programming paradigm for the first generation of clusters (networks of uniprocessor workstations). For the second generation clusters (networks of SMP workstations) a more appropriate programming paradigm is required. Therefore, a

programming paradigm called the MPIT was designed and implemented as part of the research for this thesis. The MPIT uses the MPI to communication among the workstations and the POSIX threads to run the code on multiple processors in a workstation. [Publication 5] introduces the MPIT library. The MPIT library resembles a combination of the MPI and the OpenMP [Had02]. However, it allows the programmer more control over what code the thread execute and when. Furthermore, the MPIT is not a compile-time parallelization technique unlike the OpenMP.

The following assumptions should be observed by the reader of this thesis:

1. A number of multiprocessor computer systems were used in the design, implementation and testing phases. The deployed systems included Cray T3E, DEC AlphaServer, Terttu-cluster [Kos00], and Linux cluster at the Lappeenranta University of Technology. Therefore, results achieved vary and are not always comparable. In all possible cases speedups are measured and reported to allow for comparisons.

2. The scheduling algorithms presented in this thesis assume that an existing sequential application or a new application can be run in a data-parallel fashion. The algorithms have methods designed to distribute workunits among processors that all execute the same code. Furthermore, an optimal algorithm depends on the information that indicates the computational requirements of the workunits. Thus, the algorithms need a quantitative indicator of the computation required to efficiently distribute workunits.

Chapter 2. Parallel Environments

There are a number of ways to subdivide parallel environments into categories [Kum01][Hwa98]. Perhaps, the most common categorization is based on memory architectures, which divides parallel environments into three categories: shared memory environment, distributed memory environment, and a combination thereof. For the purpose of this thesis the combination of the shared and distributed memory environments is considered to be a network of uni/multiprocessor workstations. All three environments were studied to determine their unique characteristics and features. The information retrieved from this investigation was utilized in designing and implementing application-specific scheduling algorithms and the MPIT.

2.1. Introduction to Parallel Environments

This subsection briefly introduces the three previously mentioned parallel environments. The unique characteristics of each parallel environment are explored to determine their benefits and drawbacks.

2.1.1. Shared Memory Environment

A shared memory environment is one step up from a standard single processor PC or workstation. The architectural difference of a shared memory environment compared to a workstation is in the number of processors available within the system. The shared memory environment is comprised of at least two processors. The processors are connected to each other with a network. The network further connects the processors to a global memory (terms *global* and *shared memory* are used interchangeably). Thus, the network is called a memory access network to distinguish it from other networks discussed later when the distributed memory environment and the networks of workstations are introduced. Figure 2-1 depicts a generic shared memory environment with the previously mentioned components. The main purpose of the memory access network is to provide processors with fast access to the memory. The actual speed of the network is hardware and topology dependent. A number of network topologies have been designed and implemented. Perhaps, the most common topology found in systems with a relatively small number of processors is a bus. Due to contention problems with the bus topology, large systems utilize different topologies such as meshes and hypercubes [Kum01].

A shared memory environment allows for equal access to the memory by all the processors. The memory is used for purposes other than merely storing data. The global memory provides a means for the processors to communicate with each other. In practice, the communication is similar to a memory write or read operation. The sender processor stores the data to a predefined memory location from which the target processor reads it. The use of the global memory introduces the need for synchronization. The synchronization is required to avoid a situation where more than one processor is writing to the same memory location

simultaneously. In addition, synchronization primitives can be utilized to control the execution of processors, such as to implement a barrier.

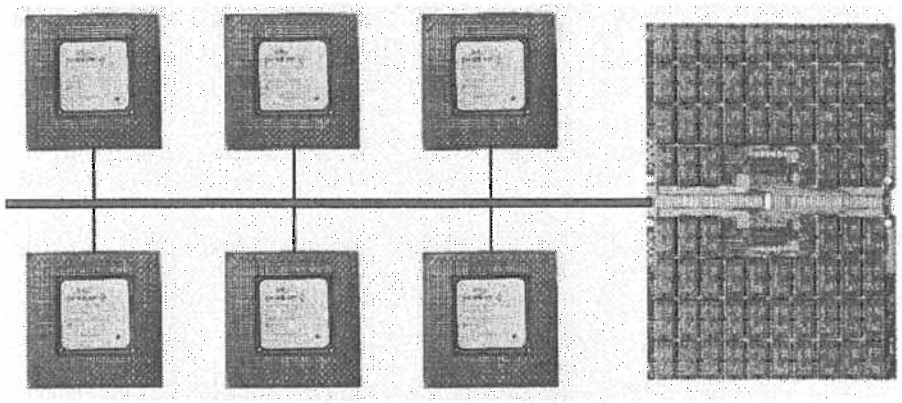


Figure 2-1. The shared memory environment.

Due to their architectural design, shared memory environments offer a relatively simple programming environment. Basically, the programmer's only concern is to handle synchronization in such a way as to avoid concurrent access to the same address in the memory. However, the problem with shared memory environments is the contention introduced by the processors to the memory access network. The processors need the network to retrieve and store data as well as instructions. The network is capable of serving a single request at a time, which introduces delays to memory access times and impacts the execution times of applications run on the processors. The contention problem is the most apparent in a single bus network (see Figure 2-1), where all processors are connected to the global memory via this bus. In order to overcome the contention problem it is possible to utilize different network topologies, and divide the global memory into smaller memories and distribute the smaller memories among a set of processors.

2.1.2. Distributed Memory Environment

The distributed memory environment solves the problem caused by the network in a shared memory environment. A distributed memory environment is depicted in Figure 2-2. The environment consists of a number of processing elements (PEs). Each PE has at least one processor, memories (RAM, cache), and a network interface. Optional hardware for a PE are additional (special) processors, cache memories, network interfaces, and I/O interfaces. The PEs are connected to each other with a high-speed network. The network is utilized by the PEs to send and receive messages. A number of network topologies have been proposed [Kum01]. Perhaps, the most common topologies are various types of meshes and hypercubes.

Since each PE has all the necessary hardware components to process data, the programming in a distributed memory environment involves the implementation of a message passing scheme. The purpose of the message passing scheme is to facilitate the exchange of data among the PEs. A memory synchronization issue similar to the one found in shared memory environments does not exist in distributed memory environment. This further simplifies programming in distributed memory environments.

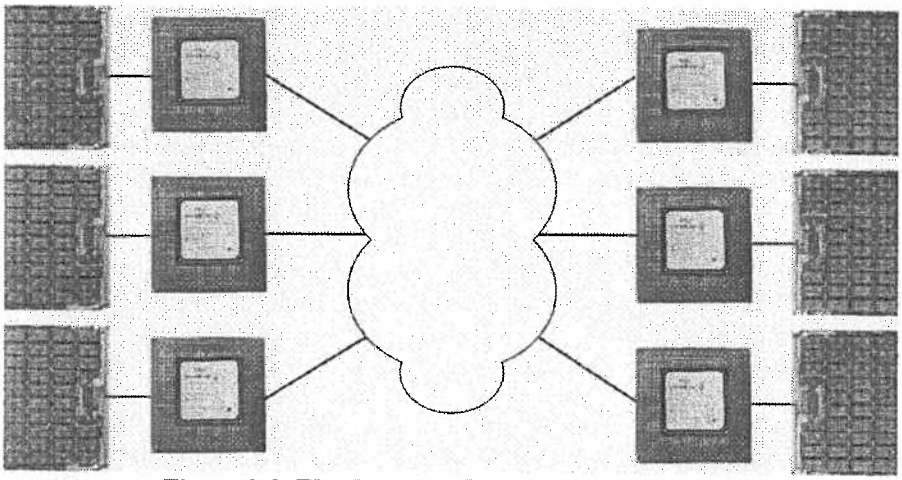


Figure 2-2. The distributed memory environment.

2.1.3. Networks of Workstations

A network of workstations is a collection of independent workstations connected to each other with a network [Sto01]. Technically, a network of single processor workstations is a distributed memory environment. However, in distributed memory environments the processing elements are normally dedicated to parallel computing and the network architecture and protocol are optimized for message passing. In a NOW environment the workstations are not necessarily dedicated, whereas the "owner" of the workstation has preference over the workstation's resources. Furthermore, the workstations can be, and usually are, heterogeneous with respect to hardware (processor speeds, amounts of memory) and software (operating systems, binary formats), whereas in a distributed memory environment the processors are usually identical. In addition, the network connecting the workstations in a NOW does not use the proprietary protocols similar to the ones used in distributed memory environments [Kat97]. In NOWs, protocols such as GM (Myrinet), Ethernet, Active Messages and other lightweight messaging protocols are commonly deployed [Chi99][Nie01][Par99].

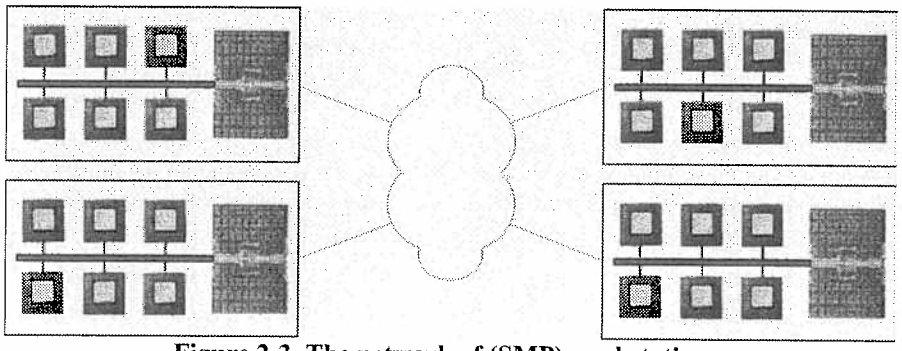


Figure 2-3. The network of (SMP) workstations.

The NOW computing offers a cost-efficient way of achieving the computing powers of supercomputers. In most cases, the cost of building a NOW is a fraction of the cost of purchasing a proprietary supercomputer with an equal number of processors. The recently

published Top 500 list of the fastest computers in the world has over 60 NOW systems [Top02]. Many of these systems are ahead of proprietary supercomputers such as IBM SP2s and Cray T3Es. An additional goal of NOW computing is to provide a powerful, low cost, common-of-the-shelf (COTS) system by utilizing available resources of workstations in office environments [Den01][Qin97][Sav99][Uth02]. The utilization of workstations in an office environment is a feasible option, since studies have shown that workstations are without work for the most of the time [Ach97]. Furthermore, due to the performances of present-day processors a casual usage of a workstation does not impose a significant load on the workstation. Therefore, the workstations are available to participate in the parallel computation even if they are occupied by their respective users.

The most recent development in parallel architectures is a combination of the shared and distributed memory environments. The architecture brings together a number of multiprocessor workstations, forming a network of symmetric multiprocessors workstations [Tan99]. The communication between processors within the same workstation utilizes the global memory, and the communication between the workstations deploys the interconnecting network. Furthermore, the maximum number of processors in such a system is not limited by the memory contention problem, as is the case in shared memory environments. This is a result of the workstations having their own memories and being connected with an interconnecting network.

2.2. Communication in Parallel Environments

All three environments previously described introduce communication. In a shared memory environment the processors utilize the global memory to pass messages to each other. In a distributed memory environment, the processors send messages to each other via an interconnecting network. In an SMP NOW communication occurs in two different steps: within and among workstations. This subsection discusses communication and its impact on applications in these environments.

2.2.1. Introduction

Applications written for the three environments require communication among the participating processors. Depending on the application, the communication occurs at certain points of computation: data is distributed at the beginning of computation, exchanged during the computation, and gathered at the end of the computation. Since the communication is considered overhead, it can have a significant impact on the performance of a parallel application.

In order to quantify the impact of communication on the performance of an application, a number of communication characteristics for each environment were identified and measured. The following characteristics were considered:

- Completion time
- Latency
- Throughput
- Protocol
- Synchronization

Completion time indicates the time from the moment a communication call is issued until the control is returned to the calling function. This includes all necessary operations taken by an operating system and/or message passing library in preparation for a message transfer. The completion time can vary drastically depending on the sending and receiving procedures. In cases where a synchronous communication operation is performed, the control does not return to the calling function until the operation has been completed. With an asynchronous communication operation, the control returns immediately after the data has been passed to the entity responsible for the message transfer. The completion times were measured for the TCP and MPI calls (synchronous and asynchronous). In the TCP, the completion time was considered to be the time spent on calling the *write* function. In the MPI, the completion times were measured for synchronous and asynchronous calls. The synchronous call did not return until a message was successfully sent or the message was copied to a system buffer, whereas the asynchronous call returned control to the caller immediately.

Latency defines the total communication time from the moment a source processor issues a send operation to the moment when the target processor retrieves the message. The latency indicates the actual communication time observed by the receiver processor. The latency is a very common performance measurement used to compare interconnecting networks. For a parallel application, latency is an important issue. Time spent on communication is time away from the actual computation, if no communication-computation overlap is achieved. The latency was measured with a number of different interconnection networks, such as three Ethernet networks and a Myrinet network, to quantify the impact of communication on parallel applications.

Throughput defines the bandwidth utilization. The bandwidth is constant for each type of a network (for example, 1000 Mbit/s for a Gigabit Ethernet network). However, throughput determines how well a message transmission can utilize the bandwidth provided by the network. In general, large messages are preferred over small messages, since larger messages can take advantage of the full bandwidth. Furthermore, the impact of the overhead introduced by the message processing at sending and receiving processors is minimal, i.e. the message setup time is a fraction of the total communication time (latency). Even though the use of large messages leads to better throughput, they are more volatile to retransmissions and congestion. Therefore, from the performance point of view, small messages are preferable instead of one large message due to the fact that the communication time matters more than the bandwidth utilization. This is especially true, if the number of messages is relatively small. However, in certain cases it can be beneficial to postpone the send operation and wait until more data is sent prior to sending a single, large message to the network. The throughput measurements were conducted in conjunction with the latency measurements for the four networks.

Protocol defines how the data (messages) are actually transported over a network. The main responsibility of a protocol is to determine how the messages are routed through the network. The topology defines the possible routes for message transmissions handled by the protocol. The protocol must be able to determine the shortest (fastest) route between the communicating processors. In a partially connected network, the protocol has to route the messages through intermediate processors. These intermediate processors may (store-and-forward routing) or may not (cut-through routing) have to participate in the communication [Kum01]. Also, the protocol has to be able to find alternative routes for messages, if the shortest route is congested or down. A secondary meaning of the term *protocol* is to define an

end-to-end transportation mechanism for the communication. The TCP and the UDP are examples of end-to-end protocols. The TCP provides a reliable transportation service, whereas the UDP is an unreliable transportation service. Both protocols have benefits and drawbacks; the TCP creates overhead by providing a reliable connection, and the UDP can lose data due to the unreliable connection. Thus, there are a number of advanced protocols implemented for message passing that try avoid problems in the TCP and the UDP [Nag99][Ste97][Wea99]. The impact of the protocol was not quantified as such, since a limited number of protocols were available. There were only Ethernet and Myrinet protocols at hand, and none of the Ethernet networks ran with the same speed as the Myrinet network.

Synchronization deals with the way the source and destination processors participate in the communication. When message passing libraries are used, the processors have to take part in the communication at some point; loose synchronization is allowed, if the sender can send a message without a matching receive operation. However, eventually the receiving processor has to receive the message. If a synchronized model is deployed, the execution of both the processors is halted while the message is sent and received; the communication reduces the level of parallelism, since during the data transfer the processors cannot process data. Message passing libraries designed primarily for proprietary supercomputers utilize a different approach to synchronization. These libraries (such as SHMEM) allow a sender to handle message passing without interfering with the execution of the destination processor [Sco96]. The sender processor is also able to process data during the communication; the overlap is possible due to asynchronous communication models [Par99]. On the other hand, synchronization can be understood to be a mechanism to control the execution of processors. Message passing libraries, such as the MPI, offer methods to stop the execution of a processor until a certain number of processors have reached the same point in their execution (barrier synchronization). In order to perform such an operation communication is required. The actual communication pattern is dependent on the message passing library.

2.2.2. Shared Memory Environment

As previously noted, the communication in a shared memory environment is normally handled through a memory access network. This communication occurs without the user's involvement. Due to the nature of the environment a different approach for investigating the characteristics of the shared memory environment was taken. Studies were made to quantify the overhead introduced by the communication. In practice, the time spent on accessing the global memory was measured. The memory access to a specific memory address has to occur in a synchronous fashion to guarantee the integrity of the stored data. A synchronization method is required to avoid a situation where two or more processors are accessing the same memory location at a time. A similar analogy applies to distributed memory environments where the synchronization takes place as an indirect result of message passing.

In order to study the significance of a synchronization method, tests were conducted on a 4-processor Pentium III Linux workstation. All the processors were connected to each other via a bus. The test application measured the time it took for a processor (thread) to obtain a mutual exclusion (mutex) lock, write data to a memory, and release the lock. The results are shown in Figure 2-4. The figure illustrates the results with and without the synchronization method. It should be noted that if it is possible for more than one thread to access the same memory location at a time, a synchronization method is required. Therefore, the results without synchronization are meant to be merely a reference to depict the overhead generated by the mutual exclusion lock.

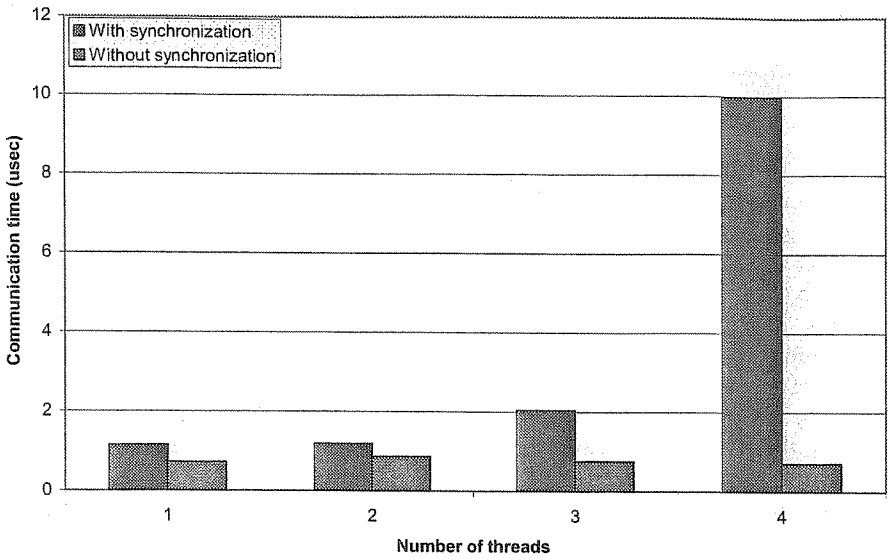


Figure 2-4. The thread communication times (averaged over all threads) with and without a synchronization method (mutual exclusion locks).

The results indicate the average write times of threads to a shared memory location. With one thread it takes approximately 1.14 microseconds to perform the synchronization steps and write the data, whereas without the synchronization the time is reduced to 0.72 microseconds. The communication time increases as more threads are used. With four threads the average communication time of 9.97 microseconds was observed. This constitutes a 9-fold increase over the test with one thread. Reasons for such a large increase in the overhead are the synchronization, thread scheduling, and cache misses. On the other hand, the communication time without the synchronization mechanism remains relatively constant when the number of threads is increased. This clearly illustrates overhead generated by the synchronization. Therefore, it is essential that the need for synchronization is minimized and thoroughly optimized.

2.2.3. Distributed Memory Environment

Communication is a significant factor in a distributed memory application, since the application must determine how and when the communication is performed. For distributed memory environments results for the completion time, latency, and throughput can be measured, since actual messages are passed from one processor to another.

The results were obtained for the TCP and MPI packets. The implementation of the MPI (*mpich*) used the TCP/IP as the message transportation protocol. Four networks were used in the tests: three Ethernet networks (10/100/1000 Mbit/s) and a Myrinet network (2 Gbit/s) [Kim01]. Ethernet is a well-established network protocol that has been widely used and studied. Myrinet is a proprietary network infrastructure that requires special network interface cards and optical interconnections. Myrinet has its own implementations of the TCP/IP stack and message passing libraries. The company that manufactures Myrinet products (Myricom) provides a reference implementation of the TCP/IP and an optimized implementation of the MPI. The TCP/IP stack and the MPI utilize the underlying network protocol called GM,

which is the native protocol for a Myrinet network [Myr00][Nie01]. It is possible to use the GM protocol directly to minimize the overheads introduced by high-level protocols and message passing libraries such as the TCP and the MPI.

Completion Time

The completion time was quantified by measuring the time a processor had to wait after issuing a send operation until the control was returned to it. Thus, the completion time indicates how long it takes for the issuing processor to complete the operation. In the TCP, a communication request included a data transfer over a socket. For the purpose of measuring the completion time it was assumed that each packet was sent on an existing socket connection; the establishment and tear down of a socket were not considered as part of the completion time. In the MPI, the completion time was measured by quantifying the time spent on calling an appropriate MPI function call. Depending on the MPI function called, the completion time varied significantly. On one hand, there is a synchronous send operation that does not return the control to the calling function until the message has been sent or copied to a system buffer. If the message is not sent or buffered immediately, the completion time can be long. On the other hand, the MPI provides an asynchronous send operation that does not wait for the message to be sent or buffered; it returns immediately after the message has been given to the MPI for transmission. Since the asynchronous operation returns the control immediately to the calling function, it does not introduce overhead regardless of the state of the receiving processor. Table 2-1 depicts the completion times for the TCP and the MPI. Two results are given for the asynchronous MPI call: the first one is from a situation where the matching receive operation has been posted (also the case with the synchronous MPI call), whereas the second one shows the completion time when no matching receive operation has been posted.

The results show that the completion time for writing to a TCP socket is substantially less than is the case in the MPI. This can be explained by the additional work done by the MPI prior to sending data to a receiving processor; the MPI has to determine the receiving processor and create an envelope for the data. The envelope contains information required by the receiving processor to retrieve the message. The results indicate that the asynchronous MPI call is approximately 20% faster than the corresponding MPI synchronous call. It should be realized that the improvement would be significantly greater, if no receiving operation had been posted when the synchronous MPI send operation was executed.

Table 2-1. The completion times (in μsecs) for the TCP and the MPI.

TCP	MPI synch.	MPI asynch.	MPI asynch., no receive operation
1.9	73.2	71.1	59.3

The MPI implementation, *mpich*, used in the tests utilizes the TCP/IP as the transmission protocol. Therefore, the TCP completion time is hidden in the MPI implementation. Finally, it is important to realize that these completion times indicate the time spent at the application level to start a communication process. Latencies discussed next show the time spent on the end-to-end communication itself.

Latency

Latencies were measured in all four networks. Figures 2-5 and 2-6 illustrate the results achieved for the TCP and the MPI, respectively. Both the figures show relatively typical performance results. With small packet sizes the overhead of the TCP and the MPI implementations cause the performance to be relatively constant. Differences in the latencies with small message sizes are explained by the overhead introduced by the protocol software (the TCP/IP stack and the MPI implementation) and the network. Once the message size grows, the characteristics of the network become more obvious.

As Figure 2-6 illustrates the latencies for the MPI are slightly higher than the latencies for the TCP. This is a result of the fact that the MPI operates on top of the TCP/IP stack. Prior to utilizing the TCP/IP stack to transmit a packet, the MPI packages the data into an envelope. The results indicate that until a message reaches a certain size (unique to each network), the communication time is practically constant. The bandwidth of the network determines the point where the message size begins to affect the communication time; a faster network can transmit larger messages with a constant time. If these constant communication times measured for the TCP and the MPI are compared, the overhead introduced by the MPI can be quantified.

Table 2-2 shows the average overhead generated by the MPI compared to the TCP. A closer look at the overhead introduced by the MPI reveals that depending on the network the overhead is incurred in different phases of communication. In the Ethernet networks the largest overhead was measured when the message size was small. Thus, the preparation of the message for transmission had a significant contribution to the overhead. In the Myrinet network, the highest overhead was measured with very large messages. This can be partly explained by the MPI library deployed. The optimized MPI implementation for the native communication mechanism (GM) of the Myrinet network could not be utilized due to technical difficulties. Therefore, the results shown are for an MPI implementation that was built on top of the TCP/IP stack and the GM.

There are some very interesting aspects in Figure 2-5 that depict the latencies for the TCP in the four networks. It seems that the Gigabit Ethernet network has larger latencies with small message sizes than the 100 Mbit/s Ethernet. This can be the result of a number of factors: internal buffering, delayed acknowledgement, or a non-optimal driver.

Table 2-2. Overhead of the MPI in percentage-wise compared to the TCP.

Network	Overhead
Ethernet (10 Mbit/s)	
Ethernet (100 Mbit/s)	39 %
Ethernet (1 Gbit/s)	49 %
Myrinet (2 Gbit/s)	110 %

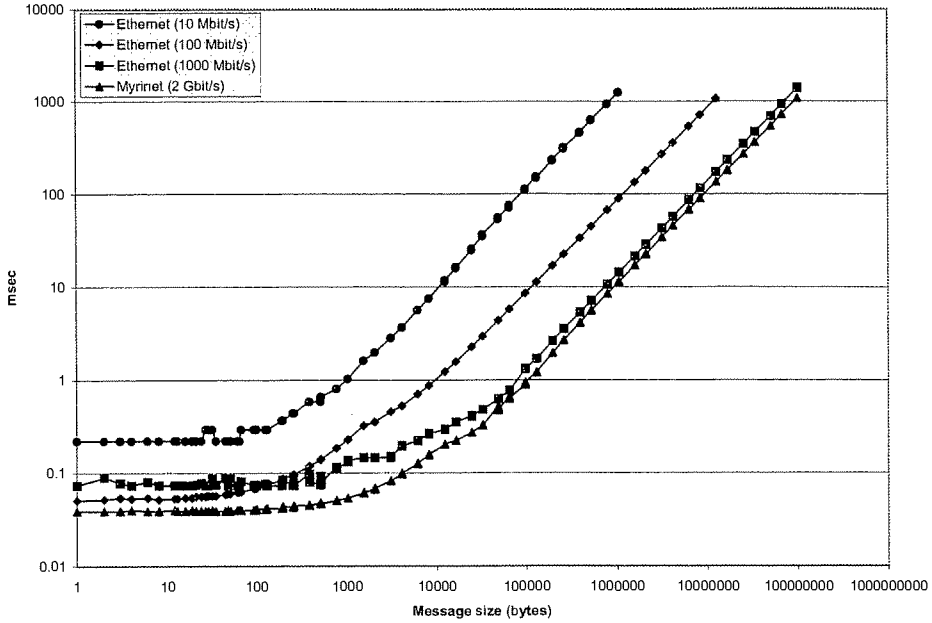


Figure 2-5. The latencies measured for the TCP.

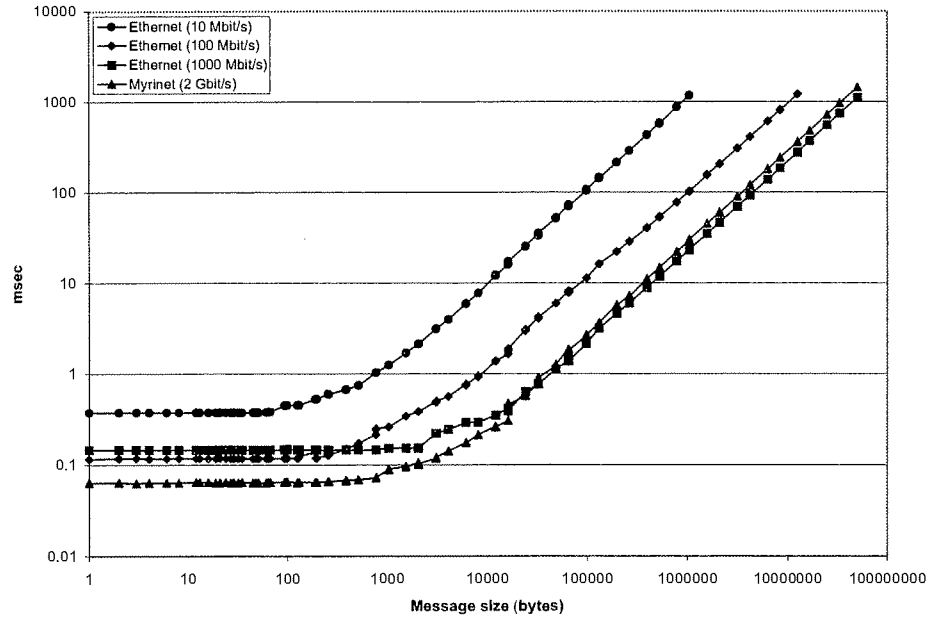


Figure 2-6. The latencies measured for the MPI.

Throughput

Figures 2-7 and 2-8 show throughputs measured for the TCP and the MPI, respectively. The results depict a situation that could already be seen from the latency figures; the TCP is capable of utilizing the available throughput substantially better than the MPI due to overhead introduced by the MPI. Furthermore, the figures illustrate that the network utilization does not necessarily reach its peak performance with the largest message size. The peak throughput message size depends on a number of factors such as protocol, buffer sizes, and network congestion. Only the 100 Mbit/s Ethernet network seems to achieve peak performance with the largest message size that was used in the tests. For all other networks the peak performance is achieved with smaller than maximum message sizes. In addition, once the peak performance has been reached the throughput starts to asymptotically reach a sustainable transfer rate. Peak throughputs with the corresponding message sizes for the TCP and the MPI are shown in Table 2-3.

Table 2-3. Peak throughputs in Mbit/s with the message sizes for the TCP and the MPI.

Network	TCP	MPI	Difference
Ethernet (10 Mbit/s)	8.5 (4 kB)	8.1 (8 kB)	5 %
Ethernet (100 Mbit/s)	89.7 (13 MB)	79.9 (8 MB)	11 %
Ethernet (1 Gbit/s)	647.4 (65 kB)	360.0 (65 kB)	44 %
Myrinet (2 Gbit/s)	840.4 (98 kB)	405.7 (16 kB)	52 %

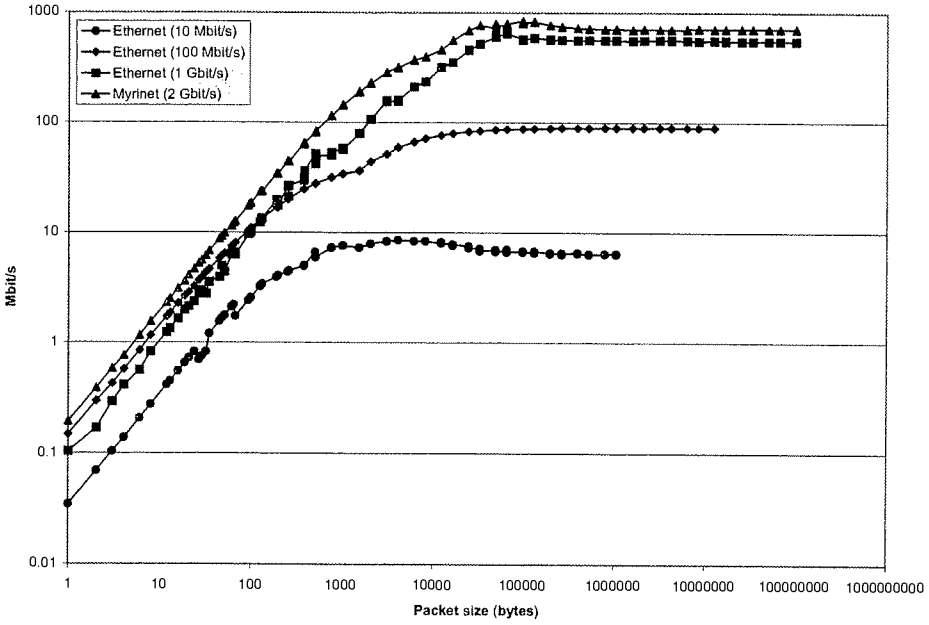


Figure 2-7. The throughputs measured for the TCP.

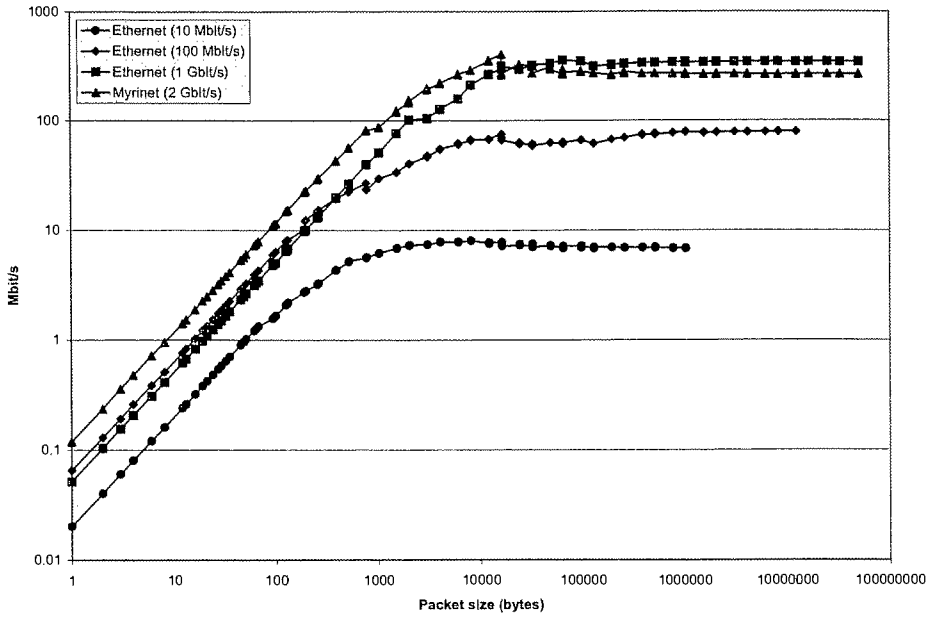


Figure 2-8. The throughputs measured for the MPI.

2.2.4. Network of Workstations

Communication is an equally important factor to consider in NOWs as it is in distributed memory environments. If a NOW consists of uniprocessor workstations, the communication scheme is similar to the one found in distributed memory environments. However, if the NOW includes workstations equipped with more than one processor, it has an impact on the communication. In an SMP NOW part of the communication takes place among the processors within the workstations [Hsi00]. This communication utilizes the shared memory as described previously. However, the communication among the workstations in a NOW still has to travel via an interconnecting network. The communication results for a shared memory environment shown in Figure 2-3 demonstrated the communication among threads in an SMP workstation. The results shown for the interconnecting networks in Figures 2-3 and 2-4 were actually measured in an SMP NOW. These figures give insight into how an SMP NOW works and what are its communication performance boundaries. Ultimately, it is left up to the programmer to decide how to implement a communication scheme within and amongst workstations.

2.3. Programming in Parallel Environments

This subsection explores programming paradigms for the three parallel environments. The most common programming paradigms are introduced for each environment and the paradigms used with the case studies in Chapter 5 are discussed in detail.

In general, each environment has its respective programming paradigms. However, there are programming paradigms for shared memory environments that use message passing. For example, implementations of the MPI exist that support communication via a global memory. On the other hand, with virtual/distributed shared memory programming paradigms it is

possible to write parallel applications for distributed memory environments as if there were a single global memory [Dre98a][Dre98b]. These cross-environment programming paradigms are not discussed further. The focus of this subsection is on the native programming paradigms for each environment.

2.3.1. Programming in the Shared Memory Environments

The obvious programming paradigm for shared memory environments is to create a set of processes that utilize a shared memory region to send and receive messages to and from each other. However, the use of processes is expensive with respect to context switching and the need to cross process boundaries for memory access. To alleviate these problems the concept of threads was introduced. Threads allow for the generation of multiple instances inside a process that run code simultaneously assuming an adequate number of processors is available [Roh96][Kle96]. There are two widely adopted thread packages: POSIX threads (pthreads) [But97] and Solaris threads [Lew96][Nor96]. The former is a standard supported by the majority of operating system vendors. The latter package is available for all SUN operating systems. Due to its operating system specific nature, the Solaris threads package is implemented on a lower level (coupled closely with the operating system) providing improved performance. However, since the Solaris thread package is not suitable for any other operating systems, the pthread package has become the thread standard for shared memory systems.

A parallel application runs inside a single process allowing all the threads, even if run on different processors, to share a memory allocated for the process. Data exchange takes place through the shared memory. Since the memory access is fast, the communication does not introduce a great deal of overhead. The main contributor to the overhead in a shared memory environment is the synchronization of the threads. The synchronization is required, since otherwise two or more threads are able to access the same memory location simultaneously. The synchronization is not implemented on a hardware level but instead it is left up to the application. The problem with the synchronization is that if multiple threads are accessing the same memory location simultaneously, only one thread is allowed to actually access the memory at a time. Thus, the other threads have to halt their execution and wait until they can access the memory.

During the design phase of a parallel application a suitable memory access model must be considered in order to minimize the synchronization overhead. With a proper design of the application, the synchronization overhead can be reduced considerably. If simultaneous memory accesses to same memory locations can be avoided, the need for synchronization is minimized, or even eliminated. Thread packages offer various methods for the synchronization [But97]:

- Mutual exclusion locks
- Condition variables
- Semaphores

Mutual exclusion (mutex) locks allow for the creation of a protected area that is executed by one thread at a time. A mutex lock must be acquired by a thread prior to entering the protected area. If the lock is held by another thread, the thread trying to acquire the lock is blocked. Once the lock is released by a thread exiting the protected area, the blocked thread

can acquire the lock and proceed with its execution. If more than one thread is waiting for the lock to be released, one of the threads is selected and allowed to proceed.

Condition variables allow a thread holding a mutex lock to sleep until it receives a signal. When a thread blocks its execution, it releases the mutex lock. The execution of the thread does not continue until it receives a signal from another thread. The signal is sent by utilizing a specific condition variable signal function call.

Semaphores are counters in the sense that they keep track of available resources. A semaphore is initialized by setting a value to it. When a thread requests a new resource it attempts to decrease the value of the semaphore. If there are resources available, and the value of the semaphore is greater than 0, the thread decreases the semaphore and proceeds with its execution. On the other hand, if no resources are available the thread is blocked until a resource becomes available. Resources become available when threads no longer require them and issue *post* operations that release the resources and increase the value of the semaphore.

It should be noted that the descriptions above merely discuss the basic functionality and usage of each thread synchronization method. The actual functionality and any possible additional features of each method are implementation dependent on thread libraries. Furthermore, there are programming paradigms, such as the OpenMP, that hide the explicit need for synchronization due to the nature of the paradigms. In the case of the OpenMP, the parallelization is performed by the compiler with the help of information provided by the programmer [Ayg99][Had02].

2.3.2. Programming in Distributed Memory Environments

The programming paradigm for distributed memory environments is called message passing. The name stems from the fact that processes send and receive messages to and from each other. For distributed memory environments two kinds of programming paradigms can be found. First, most distributed memory system vendors have implemented their own message passing libraries that are optimized for their systems [And97][Lue99][Myr00]. Second, there are portable programming paradigms that try to standardize the programming model for distributed memory environments. The most widely used portable programming model is the Message Passing Interface (MPI) [MPI95][MPI97]. The MPI provides a rich set of point-to-point and collective communication functions. It should be noted that the MPI as such is merely a standard. The most popular implementation of the standard is *mpich* by Argonne National Laboratory at the University of Chicago [Gro96]. The other implementations are LAM [Bur94], OOMPI [Squ96], and MPI++ [Kaf95]. The last two provide C++ interfaces/bindings for the MPI, whereas *mpich* and *LAM* have interfaces for C and FORTRAN. Most of the free MPI implementations are built on top of the TCP/IP stack. Due to the fact that the basic MPI implementations utilize point-to-point communication schemes to perform collective communication operations some modifications to improve the performance of communication have been discussed in literature [Bru97a][Lau97][Sis99][Skj94]. Another well-known message passing library is the Parallel Virtual Machine (PVM) [Gei96][Sun90] [Sil99][Spr01].

The popularity of the MPI and the PVM can be attributed to the easy of their use and their portability. However, there are a number of other communication libraries capable of offering the same kinds of services as the MPI and the PVM. The SHMEM (Shared memory) library

is a communication library that abstracts the communication as memory references [And97][Lue99]. The SHMEM operations allow the programmer to access the memories of other processors without interrupting the processors work. Thus, the communication does not require involvement from the processor while the memory is being read or written. This kind of scheme is generally called one-sided communication.

Active Messages is another one-sided communication paradigm [Par99]. It has a send operation, but no receive operation. The sender processor transmits the data and a handler to a receiving processor. In the receiving processor, the message is retrieved from the network by a dedicated process that creates a new thread to process the data. The action taken to process the data is defined by the handler. The code executed by the thread to process the data has to be implemented by the programmer. The benefit of Active Messages is its capability of not interrupting the work of the receiving processor when messages arrive and are processed. In addition, the design of Active Messages thrives to minimize the number of memory copies required to pass the data to the thread for execution. Since the thread is created by the process handling the reception of the messages, a minimum number of copy operations take place prior to the processing of the data. This leads to the expedient processing of messages received by a workstation.

2.3.3. Programming in Networks of Workstations

In order to run parallel applications in a NOW, a message passing library is often the only software required. The same software that is used in the distributed memory environments is suitable for the NOWs. However, due to the fact that most NOWs are heterogeneous with respect to operating systems and/or hardware, common message passing libraries (especially the MPI and the PVM) have become the libraries of choice.

The common message passing libraries are suitable as such for the first generation of NOWs. A first generation NOW is a system where each workstation has a single processor. On the other hand, a second generation NOW is comprised of workstations that have more than one processor. Granted, popular message passing libraries such as the MPI and the PVM are still suitable programming paradigms. However, to fully utilize the available resources in a second generation NOW, a programming paradigm that combines message passing and threads is required [Hen00]. The combination of message passing and threads introduces complexity to the programming paradigm. Therefore, one of the goals of this thesis is to present an MPI-like programming paradigm for writing parallel application on second generation NOWs. The paradigm developed is discussed in Chapter 4.

The significance of scheduling is emphasized in second generation NOW environments, since the work balance has to be achieved globally among workstations, and locally among processors. The global scheduling requires communication over the interconnecting network, whereas local scheduling introduces communication through the memory access network. The optimal balance between the usage of the interconnecting network and the memory access network is essential, since the communication over the interconnecting network is considerably slower compared to the memory access network. Another factor that complicates scheduling in a non-dedicated NOW is loads in the workstations generated by their respective owners. Depending on the usage of the workstations, the loads may fluctuate drastically. A similar problem occurs, if an SMP NOW consists of heterogeneous workstations; in that case, a scheduling algorithm has to take into consideration the computing capabilities of the workstations while scheduling workunits. Scheduling issues are

discussed in detail in Chapter 3 where the design and implementation of data-parallel applications are explored.

2.4. Discussion

Each of the three environments has its advantages and drawbacks; no single environment is superior to the other two. The environments differ not only from the memory architecture point of view, but the programming paradigms used to write applications for the environments differ as well. The memory architecture also has an impact on the design of an application such as how to efficiently handle communication and scheduling. The communication is an integral part of parallel computing especially in the distributed memory environments and NOWs. The studies conducted indicated that the communication had a significant impact on the execution of a parallel application. In general, it can be safely stated that the number of communication operations and the amount of data transferred should be minimized to lessen the impact of communication. Therefore, solutions to optimize the communication should be explored on a software level as well as hardware (not discussed in this thesis) level. Optimization on the software level includes implementing efficient parallel applications with proper programming paradigms, and utilizing appropriate scheduling algorithms. Chapter 3 discusses how to write data-parallel applications in the three environments. Furthermore, the chapter introduces three scheduling algorithms with the application-specific enhancements, which take advantage of information concerning the application being parallelized and the parallel environment. The programming paradigms are discussed further in Chapter 4 where a programming model for SMP NOWs is presented.

Chapter 3. Utilizing the Parallel Environments

This chapter describes how to take advantage of the three previously described environments while designing and implementing parallel applications and scheduling algorithms.

3.1. Introduction

In order to design and implement efficient parallel applications two main concepts must be understood within the scope of this thesis:

- Computation
- Communication

The first concept, computation, is the actual work done by processors. While there are numerous parallel programming and computation methods [Ski98][Wal98], two programming methods can be identified with respect to how workload is distributed among processors [Gan96][Răd01]. In the functional (task) parallel method, the computation is performed so that each processor is assigned a specific task [Liu99][Nor93]. The processor performs the same operations to all the data it receives. In the data-parallel method all processors execute the same code with different data [Kin88][Wu98]. For optimal performance, each processor must have an adequate amount of work to process. In a dedicated homogeneous system, this indicates that each processor has an equal amount of work, whereas in a heterogeneous system the amount of work should reflect the capabilities of the processors. It is the responsibility of a scheduling algorithm to optimize work balance among the processors. In addition to scheduling there are operations that need to be performed by a parallel application that do not exist in a sequential application. These kinds of operations are initialization of a parallel environment, synchronization, and communication. Naturally, the performance improvement achieved through parallel computing must be greater than the negative impact of the overhead incurred.

The second concept, communication, was discussed in Chapter 2. The results achieved showed that with faster networks the impact of a single communication operation could be minimized. However, a parallel application often needs to perform more than one communication operation. Furthermore, focusing only on latency does not give the whole picture. A parallel application introduces additional factors on the software level that affect the overall communication time. There are a number of instances where communication is required by a parallel application:

- Scheduling
- Data exchange during computation
- Synchronization

- Initialization and result gathering

The purpose of *scheduling* is to assign work to processors. The assignment usually includes the transfer of workunits to processors. Depending on the scheduling algorithm the number and size of messages may vary drastically. It should be noted that scheduling can take place either at the beginning or throughout the execution of the application. The impact of communication caused by scheduling can vary greatly depending on when it occurs.

Data exchange during the computation takes place when processors need data from each other; it is quite common for data to be exchanged during a computation phase. However, in order to perform a communication operation a processor has to stop executing the actual code and start the operation. Therefore, each communication operation contributes to the overhead in more than one way. First, the sending processor has to stop its work and perform the communication operation. Second, the receiving processor has to stop its work and receive the message. So clearly, the communication during a computation phase should be minimized. This can be achieved by reducing the data dependencies between the data held on difference processors. Another method of minimizing the communication's impact on the overall execution time of an application is to optimize the communication with prior knowledge of the communication patterns of the application [Isl97].

Synchronization is required to control the execution of processors. In shared memory environments, synchronization is used to protect areas of code from concurrent access or to control the execution of threads (barrier synchronization). In distributed memory environments synchronization is utilized to coordinate the execution of processors in different workstations. The same synchronization scheme applies to networks of workstations, whereas with networks of multiprocessor workstations, both the aforementioned synchronization schemes are applicable. All synchronization schemes introduce overhead in the form of lost computation; processors either have to wait for access to a protected code or until all of them have entered a barrier.

Initialization and result gathering are performed at the beginning and end of computation, respectively. During the initialization phase, processors can send and receive data regarding the environment, such as the number of processors and rank of each processor. The result gathering is usually performed to allow a single processor to collect results from other processors and to produce final results. Almost all message passing libraries include functions to gather data to one or more processors. In addition, since the data gathering is performed after the actual computation phase, its impact on the overall performance of the application is not as significant as it would be if it occurred during the parallel computation phase.

Scheduling was mentioned in both cases when computation and communication were discussed. In fact, scheduling (work distribution) is one of the key elements of efficient parallel computing. Its goal is to distribute a workload equally and effectively among available processors. In theory, scheduling is a relatively simple operation. Unfortunately, due to a number of impacting factors scheduling has proven to be a very complex (NP-hard) and time-consuming operation. The remainder of this chapter discusses the design, implementation and performance issues of scheduling in general and describes the scheduling algorithms developed.

3.2. Scheduling

In order to utilize more than one processor an application requires a scheduling algorithm to distribute the workload among processors effectively and efficiently. The ultimate goal of a scheduling algorithm is to maximize the performance of the processors by optimizing their utilization whilst minimizing overhead. Thus, an optimal situation would be one where all processors had an equal amount of work with respect to their processing capabilities and the amount of communication was non-existent or very small. The first requirement indicates that the numbers of workunits or the amount of work on processors does not have to be equal, and should not be equal, if the processors are not identical. A heterogeneous system introduces additional complexity to a scheduling algorithm due to the different computing capacities of the processors/workstations. The differences in the computing capacities do not necessarily imply that the processors are dissimilar; it is possible that some processors have more load than others, thus limiting their computing capacity. Communication is mainly caused by dependencies in the workload. Therefore, a scheduling algorithm should be able to identify and locate workunits with dependencies that are in close proximity (optimally on the same processor). In practice, the scheduling algorithm has to make a compromise between the loads on the processors and the amount of communication.

The actual distribution of workunits requires communication, since the workunits have to be sent to processors. A scheduling algorithm has to decide on a method for distributing the workunits; it can either send all the workunits to all processors, send only the workunits meant for a particular processor, or send indexes to the workunits to all processors. The last option requires that the processors acquire the workunits prior to the execution of the scheduling algorithm. For example, the NFS can be utilized to allow all processors to access the same set of files that contain the workunits [Publication 6].

As seen, scheduling is not, by any means, a simple task. There is no single algorithm that is able to produce optimal results for all applications in all environments. Each application and environment imposes different requirements for optimal performance and, therefore, for a scheduling algorithm. For example, in an environment where communication is a significant contributor to overhead, a scheduling algorithm that optimizes the communication should be developed. On the other hand, in an environment where the processors are heterogeneous, a scheduling algorithm that optimizes the work distribution among the processors based on the computing capabilities of the processors is favourable.

3.2.1. Related Work

Scheduling algorithms can be divided into two groups based on their operation models: static and dynamic [Bru97b][Pla94]. A static algorithm allocates workload to processors prior to the actual computation phase and no further scheduling takes place [Ros91][Tan00]. In other words, a static scheduling algorithm is an algorithm that does not receive more workunits to distribute during the computation phase. Although static algorithms compromise the optimal utilization of processors, they introduce minimal overhead. However, since static scheduling algorithms do not get executed during the computation phase, they are unable to notice and correct imbalances on processors caused by poor initial scheduling decisions or changes in load conditions.

Dynamic scheduling (load balancing) algorithms distribute the workload among processors before and during the computation phase by considering the computational requirements of

workunits and/or the current loads on processors [Ben00a][Bre99][Lei99][Ren01]. So by definition a dynamic load balancing algorithm is an algorithm that can cope with workunits that are created during the computation. For the purpose of this thesis, two operation models for dynamic scheduling algorithms are identified. The first model includes scheduling algorithms that distribute workunits as they enter the system. The algorithms do not have information about all the workunits and their computational requirements. The scheduling decisions are based on processors' loads at the moment when the workunit enters the system. Methods of measuring the system load and its effect on scheduling have been studied in literature [Das97][Kul00][Li98][Mey97]. The second operation model consists of dynamic scheduling algorithms that implement a client-server approach. The distribution of workunits occurs according to requests sent by client processors to the server (master) processor. The server processor replies to each request by sending one or more workunits to the client. The second model is preferred, because although it introduces additional overhead in the form of communication, it does not need constant load updates. The distribution of workunits occurs based on the availability and computational capacity of the processors; a processor without work makes a request for a new workunit, whereas a processor handling a workunit or occupied by another process (from another user) does not request more work until it becomes idle. The better utilization of processors and improved work balance comes with a price. Dynamic scheduling algorithms cause more overhead due to the increased communication, synchronization, and possible bottlenecks caused by the server processor. However, dynamic scheduling algorithms can substantially increase the performance of a parallel application, if implemented wisely. The more optimal utilization of processors leads to more equal work balances among the processors and decreased execution times. Dynamic scheduling algorithms have been introduced in literature for shared memory environments [Gan00][Soh97], and distributed memory environments and NOWs [Ben00a][Cav01][Cor99] [Sch95][Soh96][Tha01][Zak97][Zha00].

Algorithms for distributed memory systems normally assume that the impact of an interconnecting network is minimal or non-existent. The focus of these algorithms is to optimize the utilization of available processors. In general, scheduling algorithms for networks of workstations consider the cost of communication due to its possibly significant effect on performance [Pra97]. Scheduling decisions are made by optimizing communication and resource utilization, and by minimizing overhead generated by the scheduling algorithm.

Dynamic scheduling algorithms need information about the application and the system in which they are run. The application defines the characteristics of the workunits to be distributed, whereas the system load indicates suitable target processors for work distribution. Studies have been conducted that address the utilization of application characteristics in the scope of scheduling processes (e.g. [DaS01][Mac00][Sub02][Tan00]) or in particular applications (e.g. [Bar99][Cat01][Cha02][Jon00][Mic01][Ris02]). These studies explore the benefits gained from extracting specific information about applications; using that information the scheduling algorithms are optimized. The scheduling algorithms developed in this thesis focus on considering the generic characteristics of workunits in order to develop universal scheduling algorithms. The algorithms do not take system load into consideration. However, one of the algorithms indirectly incorporates the system load into its scheduling decisions.

There is a special group of dynamic scheduling algorithms that consider the work distribution only among their nearest neighbours [Els00][Gho99]. These algorithms allow the processors

to send and receive workunits to and from their topologically neighbouring processors. This approach minimizes the communication cost for two reasons: (1) The number of links a workunit needs to traverse to reach the destination processor is always 1. (2) No more than two workunits are transmitted over any given (full duplex) link at a time. However, it is important to realize that for the nearest neighbouring scheduling algorithm to deliver good performance, the communicating processors have to be topologically next to each other and use a dedicated communication channel. For example, two workstations, even if there are physically adjacent to each other, cannot take advantage of the nearest neighbour approach, if they are connected via Ethernet or any kind of bus network. The disadvantage of such algorithms is their inability to react to drastic changes in the loads of processors, especially if the system has a large number of processors.

A second special group of dynamic scheduling algorithms is comprised of algorithms that distribute workunits randomly among processors [Adl95][Ber99][Gho94][Mit97]. These algorithms are based on statistical models and are able to provide estimations of maximum loads on processors. The suitability of most random load balancing algorithms is limited to cases where the workunits are equal in size. Furthermore, the algorithms assume that the processors are homogeneous and have identical loads.

Some very unique and theoretical approaches to dynamic load balancing can be found in literature. [Hui99] presents a mathematic model with a hydrodynamic approach. Processors are considered cylinders, the diameters and heights of which define their computing capabilities and loads. On the other hand, scheduling algorithms for loop-level parallelism are discussed in [Lim99][Muk99]. These models operate on a lower level than other previously described scheduling algorithms by distributing parts of loops to processors.

In addition to numerous studies of scheduling algorithms, tools for performing dynamic scheduling have been developed which aim to lower the threshold to enable efficient parallel computing [Dev00][Ger91][Hen98]. In [Wil96][Wil98] an automated load balancing model is presented. The model tries to estimate the computational requirements of workunits and assigns the workunits with the most work to processors that have the lightest loads. This approach is very close to the one presented in this thesis. However, the difference is in how the workunits are actually distributed. The algorithm in [Wil96] assumes that loads on processors are known by the load balancing algorithm, whereas the algorithm developed by the author does not require any knowledge of the loads on processors.

Static or dynamic scheduling is either conducted in a distributed or a centralized fashion. A distributed algorithm is executed by all participating processors. In general, the algorithm produces a workload for the executing processor. However, it is possible that the algorithm has to inadvertently generate workloads for each processor in producing a workload for the executing processor. The benefit of decentralizing the scheduling operation is to allow for the better utilization of resources; no processor is without work even during the scheduling phase. In addition, scheduling is faster if each processor does its part rather than a single processor performing all scheduling operations. On the other hand, the drawback of the distributed scheduling model is the communication overhead. In order for the processors to perform scheduling they need information about the workloads and loads on other processors. Thus, one processor can distribute the workload information (one-to-all broadcast) among the processors, whereas each processor has to send information about its load to all other processors (all-to-all broadcast). Furthermore, if more workunits are generated during the

computation phase, the distributed algorithm introduces more communication; the processor handling the new workunit has to determine which processor it will send the workunit to by first polling information about the loads on the workstations and then sending the workunit to the assigned processor for execution.

A centralized algorithm is executed on a master processor that computes workloads for all processors. The master processor can participate in the parallel computation phase, if so required. The centralized scheduling algorithm can be either static or dynamic. In the case of a static scheduling algorithm, the dedicated processor performs a one time distribution of the workunits among the processors based on information obtained from the workunits and/or its knowledge of the load situation. With a dynamic algorithm the situation is identical to the dynamic scheduling in distributed algorithms. The workunits are distributed in two ways: according to the workload information and/or the current load situation on processors, or by requests from processors.

If the master processor does not participate in the computation, it can monitor the load situation on processors while they process data. If a dynamic scheduling algorithm is deployed, the master processor can improve the work balance by migrating workunit(s) from an overloaded processor to a processor with less work [Boy02][Cru01][Hey98][Wil98]. The benefit of the workunit migration is the increased utilization of processors, which in turn results in better speedups, although all actions performed by the master processor require communication. Therefore, the implementation of the scheduling algorithm has to be carefully designed so that the impact of communication does not negatively affect the overall performance of the application. The scheduling algorithms designed and implemented as part of this thesis do not consider load migration.

3.2.2. Requirements for Optimal Scheduling

Four issues can be identified and should be considered while designing and implementing a scheduling algorithm:

- Application
- System architecture
- Interconnecting network
- Other characteristics

Application defines the workload, and the workunits. It also provides a means to estimate or measure the computational requirements of the workunits for more accurate work distribution. The workload also indicates any possible dependencies between the workunits and their degree of dependency. In some cases, parallelization of an existing sequential application can prove to be very difficult due to the complicated structure of the application, which causes numerous dependencies. This is usually the case when a sequential application has been designed and implemented without any consideration to parallelization.

An application can provide additional information for a scheduling algorithm [Gri94]. This information can be very helpful, if the workload is generated dynamically and/or workunits provide information about their computational requirements. It is essential that a scheduling algorithm is capable of adapting to changes in the workload and utilizing information available about workunits. Furthermore, if more workunits are generated during the execution

of an application, a scheduling algorithm has to be able to distribute these workunits like any other workunits. The possibility of extracting information about the workunits brings up an interesting area of research. If a scheduling algorithm is provided with more accurate information about the workunits than, say, just the total number of workunits, could the scheduling algorithm run better and produce better results. The following subsections discuss scheduling with application-specific information extracted during the runtime of an application.

System architecture defines the parallel environment including processors and memories. First and foremost, the processor layout has a significant impact on the design and implementation of a scheduling algorithm as well as on its functionality. The layout specifies how processors and memories are coupled. Next, possible heterogeneous features in the system affect the scheduling algorithm [Ben00b]. It is not impossible for a distributed memory system to have processors with different processing capabilities. On the other hand, this is a highly likely situation in a network of workstations that has been built from workstations in an office environment for example. Heterogeneity introduces further complexity to a scheduling algorithm. The algorithm has to take the capabilities of the processors into account to compensate for the work distribution. In order to perform optimally the algorithm has to know the relative powers of processors during the actual scheduling phase. This requires that the algorithm have knowledge of the characteristics of all processors participating in the computation. However, obtaining the necessary information and processing it, is not always possible. Thus, work distribution is considered to be substantially more difficult in heterogeneous environments than in homogeneous environments. However, one of the scheduling algorithms developed in this thesis proposes a solution for scheduling in heterogeneous environments.

Interconnecting network affects the design and implementation of a scheduling algorithm indirectly. Since a network (in conjunction with other hardware such as a network interface card) defines the time spent on communication, a scheduling algorithm has to adapt to the speed of the network [Cru01][Fer01][Kai99]. There are different characteristics of a network that impact communication; with large data sets the network speed (throughput) can become a limiting factor, whereas with small data sets the completion time and latencies can dominate communication times. Very small data sets can be common, since work distribution and synchronization messages are usually quite small. In general, if the number of messages sent by a scheduling algorithm is minimized the impact of the network on the design and implementation of a scheduling algorithm is miniscule.

Other characteristics are something that cannot be predicted beforehand, but the scheduling algorithm should be able to react to them. Perhaps, the most significant characteristic from a scheduling algorithm perspective is load fluctuations on processors participating in the computation. It is possible that other users are simultaneously using some of the processors. In a time-share operating system a time slice is allocated for each application. Assuming the task priorities are identical, the task that is part of a parallel application has to share the processor with other tasks. The execution of this task is slowed down by a factor of $n - 1$, where n is the number of running (active) tasks on the processor. If such a situation occurs on even one processor, it increases the total execution time of the parallel application considerably. Therefore, a scheduling algorithm either has to know loads on processors prior to distributing the workload or has to adapt to load changes dynamically. Both approaches require that the scheduling algorithm have up-to-date information about the load situation on

the processors. The latter approach assumes that the scheduling algorithm is capable of migrating workunits from a heavily loaded processor to ones that are less loaded. The job of the scheduling algorithm becomes slightly easier, if the processors participating in the parallel computation are dedicated for one user at a time.

At best, a scheduling algorithm is capable of incorporating each of the four characteristics. However, in most cases such a generic algorithm would be very complex and introduce an excessive amount of overhead. Since the ultimate goal of a scheduling algorithm is to generate optimal load balance without introducing any overhead, it is prudent to investigate a mechanism to further improve scheduling mechanisms. Thus, application-specific scheduling is introduced which tries to take into account some of the characteristics discussed earlier in order to optimize the performance of a parallel application.

3.3. Application-Specific Scheduling

An *application-specific scheduling algorithm* is a term given to an algorithm that incorporates information extracted from or provided by an application during its execution to optimize scheduling decisions.

3.3.1. Introduction

A scheduling algorithm requires accurate information about workunits to achieve the most optimal work distribution. In conjunction with the workunits, the scheduling algorithm needs information about the computational requirements of the workunits. This information is specific to an application and its workload. Hence, the name for the scheduling method. With the additional information, the capabilities of a scheduling algorithm are increased substantially. The scheduling algorithm can make decisions based not only on the number of workunits but also on the computational requirements of the workunits. The scheduling algorithms developed in this thesis consider only the application-specific information when distributing the workunits. Operations such as querying information about loads on processors to further refine the scheduling decisions are not taken into consideration [Dai00].

Generic application-specific scheduling has not been studied extensively in literature. The research effort that comes closest to what is proposed in this thesis can be found in [Ber96]. In fact, the authors use the term application-level scheduling while introducing a framework for scheduling agents. These agents operate between an application and a parallel environment to determine an optimal set of processors on which to execute the application. The agents are provided with information about the application as well as the parallel environment. The user gives the application information to the agent, whereas an external application provides the environment information. The ultimate purpose of the work done by [Ber96] is to create a number of scheduling agents that are suitable for particular applications. This approach to application-specific scheduling is a more structured and high-level approach than the one presented in this thesis.

Three generic scheduling algorithms have been enhanced to make use of the application-specific information [Publication 4]. The algorithms are generic in the sense that they do not dictate what the application-specific information is and how it is extracted. It is enough for the algorithms to be able to compare and order the workunits based on their computational requirements. Furthermore, one of the algorithms offers a solution to the load situation

problem discussed briefly in the previous subsection; the algorithm (Algorithm 3) eliminates the need for constant load updates.

The following assumptions are made with respect to the design and implementation of the three scheduling algorithms:

- The workload has been partitioned into workunits of various sizes
- The size of a workunit determines its computation requirements.
- The workunits are randomly stored in a data structure that allows each workunit to be uniquely addressed.
- The sizes of the workunits are uniformly distributed.

Note that no performance results are given in conjunction with the descriptions of the algorithms. The results of the algorithms are presented with the case studies in Chapter 5 and [Publication 4].

3.3.2. Algorithm 1

The first algorithm (Algorithm 1) is depicted in Figure 3-1. The generic version of the algorithm distributes workunits among processors by allocating an equal number of workunits for each processor. In order to perform such an operation the scheduling algorithm only needs to know the total number of workunits. This information is not considered application-specific information, since by distinguishing the workunits, the number of the workunits can be derived. All processors execute the algorithm concurrently. Each processor determines the number of workunits and which workunits it is supposed to process.

1. Calculate an average number of workunits per processor ($\bar{u}$).
2. Assign the average number of workunits to processors, so that the first processor receives the first $\bar{u}$ workunits, and so on.
3. Assign the last r processors an additional workunit, if the average number of workunits calculated in Step 1 had a remainder r ($r > 0$).

Figure 3-1. The generic scheduling algorithm 1.

The application-specific version of Algorithm 1 incorporates information about the workunits into the scheduling procedure. The algorithm is provided with information about the computational requirements of the workunits. The algorithm, shown in Figure 3-2, first computes the average number of work per processor. Then, each processor is assigned the average amount of work to process, regardless of how many workunits are assigned. In fact, the distribution is optimized so that the amount of work assigned to each processor (except the last one, which gets all remaining workunits) can either be more or less than the average. An additional workunit may be assigned to a processor even if it brings the amount of assigned work over the average, as long as the then increased amount of work assigned is closer to the average amount than before. Since the algorithm can assign more than the

average amount of work to processors, the possibility that the last processor has substantially more work than all other processors is minimized. The scheduling procedure requires slightly more time due to the extra computation introduced by the algorithm; the algorithm has to calculate the total amount of work and the average amount of work. Furthermore, the design of the algorithm dictates that each processor executing the algorithm must produce workload distributions for all processors that have a lower rank than the executing processor. However, the application-specific algorithm is capable of distributing the workload more equally than the generic algorithm.

1. Compute the total amount of work by adding the sizes of workunits together.
2. Calculate the average amount of work per processor ($\bar{u}$).
3. Assign the workunits to processors (except the last one) so that each processor is assigned with the average amount of work.
4. Assign all the remaining workunits to the last processor.

Figure 3-2. The application-specific algorithm 1.

3.3.3. Algorithm 2

The second algorithm (Algorithm 2), depicted in Figure 3-3, distributes workunits among processors based on a statistical assumption. It is assumed that workunits of various sizes are distributed equally in the data structure that contains all the workunits. Therefore, the generic version of Algorithm 2 selects workunits to process according to the following procedure. First, each processor selects the first corner to be the jth corner, where j is the rank of the processor. Second, the processors select workunits by adding p to j, where p is the number of processors, until j is greater than the number of workunits. The algorithm minimizes the possibility (based on statistical assumption) that one processor receives all the large workunits. The problem with the generic version of Algorithm 2 is that it does not take into consideration the real computational requirements of the workunits.

1. Assign the first workunits to each processor from the workunit data structure starting at an index specified by its rank.
2. Assign the remaining workunits to processors starting from the initial workunit at an interval of p workunits, where p is the number of processors.

Figure 3-3. The generic scheduling algorithm 2.

The application-specific scheduling algorithm behaves in a manner similar to the generic algorithm in terms of how the workunits are selected by the processors. However, the actual computational requirements of the workunits are determined with the help of the application-specific information. In addition, the workunits are sorted in descending order according to

the amount of work they require. When the scheduling algorithm is executed by all processors, they select the most computational intensive workunits first. This has two benefits: first, the amount of work on processors is optimized with a small amount of overhead (sorting), second, the time processors have to wait for the last processor to finish its work should be minimized. Since all the processors start by processing large workunits, the last workunits to be processed are very small, and quick to process. Figure 3-4 shows application-specific scheduling algorithm 2.

1. Sort the workunits in descending order by their sizes.
2. Assign the first workunits to each processor from the workunit data structure at the index specified by its rank.
3. Assign the remaining workunits to processors starting from the initial workunit at an interval of p workunits, where p is the number of processors.

Figure 3-4. The application-specific scheduling algorithm 2.

3.3.4. Algorithm 3

The third algorithm (Algorithm 3,WorkPool algorithm) is based on the concept of a workpool or processor farm [Fle99][Wag97]; a single processor acts as a server (words *server* and *master* are used interchangeably) distributing workunits one at a time to client processors as per their requests. The server processor does not, necessarily, participate in the parallel computation. The responsibilities of the server processor are to receive work requests from client processors, and to reply to them.

The generic algorithm dynamically assigns workunits to client processors in no particular order. Thus, the algorithm does not consider the computational requirements of workunits; the workunits are distributed in the order in which they are stored. Figure 3-5 shows the generic scheduling algorithm 3 for the master processor. The algorithm for the client processor is depicted in Figure 3-6.

1. Wait for a request from a client processor.
2. If unprocessed workunits exist, send one workunit as a reply to the client processor, and jump to Step 1.
3. If no unprocessed workunit exist, send the end-of-computation message as a reply to the client processor.
4. If the end-of-computation message sent to all client processors, exit.

Figure 3-5. The generic scheduling algorithm 3 (server processor).

Results (see Chapter 5 and [Publication 4]) show that the generic version of algorithm 3 is very efficient and is able to produce good load balance results in certain cases. However, the algorithm is not able to produce the most optimal results unless the workunits require an equal amount of processing time.

1. Send a request to the server processor.
2. Process the workunit received from the server processor, if not an end-of-computation message.
3. Jump to Step 1, if data was not an end-of-computation message.

Figure 3-6. The generic scheduling algorithm 3 (client processor).

Since the workunits are not normally identical, a more sophisticated version of the algorithm was developed. The application-specific version of the algorithm takes into account the actual computation requirements of the workunits. In addition, the workunits are distributed among the processors so that the more complex workunits are processed first. Figure 3-7 illustrates the application-specific algorithm for the server processor. The algorithm is divided into two parts. The first part consists of Step 1. The server processor performs the step before serving any requests from client processors. In Step 1, the workunits are sorted in descending order based on their computational requirements. The second part of the algorithm includes Steps 2-5. This portion of the algorithm is performed during the parallel computation phase. In Step 2, the server waits for requests from client processors. A request indicates to the server that a client processor is out of work. The server assigns the requesting processor a workunit to process in Step 3, unless all workunits have been processed. If there are no workunits to assign, the server sends a special message back to the client processor in Step 5. The special message indicates to the client processor that it should terminate its execution. The server processor terminates the execution of the algorithm once all the workunits have been assigned and all the client processors have been sent the special message indicating that no more workunits are to be processed. The algorithm for the client processors does not differ from the one presented for the generic scheduling algorithm in Figure 3-6.

1 Sort workunits based on their computational requirements.
2 Wait for a request from a client processor
3 If unprocessed workunits exist, send the largest workunit as a reply to the client processor, and jump to Step 2.
4 If no unprocessed workunit exist, send the end-of-computation message as a reply to the client processor.
5 If the end-of-computation message sent to all client processors, exit.

Figure 3-7. The generic scheduling algorithm 3 (server processor).

The algorithm is capable of distributing workunits among processors in an optimal way without complex and time-consuming work distribution computations. Scheduling occurs by the client processors, which request more work after completing their current work. Furthermore, since workunits are sorted in descending order, the ones requiring the most work are processed first. This, in turn, reduces the time processors have to wait at the end of the computation phase, since the processors should finish their last workunits more closely together.

The algorithm is designed to work with a data-parallel application where workunits are known prior to the computation phase. For the generic algorithm 3, information about the number of workunits is adequate. However, the application-specific algorithm requires additional information about the workunits. The algorithm either has to determine the computation requirements of the workunits or this information has to be provided by the programmer. For example, in the case of the GSM network simulator (See Chapter 5) computational requirements of the workunits can be determined at runtime by examining the workunits.

The only increase in the complexity of the application-specific scheduling algorithm 3 over the generic algorithm is the sorting operation. A number of fast sorting algorithms exist, thus the impact of this operation is minimal. Furthermore, even if the application-specific scheduling algorithm takes a little longer to execute, the improved work balance achieved compensates for the overhead introduced.

3.3.5. Implementation Issues

All three scheduling algorithms were implemented to quantify their performance and impact on work balance. The results achieved are shown and discussed in Chapter 5. The results indicate that the WorkPool scheduling algorithm performed considerably better than the other two algorithms. Note that no hardware related optimizations were considered, since the algorithms were designed to run in any parallel environment. Due to the superior performance of the WorkPool scheduling algorithm, implementation issues of the other two scheduling algorithms are not considered.

The implementation of the applications-specific WorkPool algorithm has two variants: one for shared memory environments, and another for distributed memory environments and networks of workstations. The following subsections discuss the special features of the two implementations.

Shared Memory Environment

The implementation of the WorkPool scheduling algorithm in a shared memory environment does not require explicit communication. Threads do not need to request a new workunit per se, they can simply fetch a new workunit by utilizing the shared memory. A synchronization method needs to be put into place to provide threads with unique workunits. The synchronization method guarantees exclusive access for a thread to obtain and increase an index to the next unprocessed workunit. The protected area of code consists only of assignment and addition operations. Thus, the synchronization does not impose a great deal of overhead. Naturally, the synchronization primitives (mutex lock and unlock) generate overhead but their impact on the total execution time is minimal.

Distributed Memory Environment and Networks of Workstations

In the implementation for distributed memory environments and networks of workstations, one processor is dedicated to act as a server processor. The responsibility of the server processor is to assign workunits to other processors. The client processors send requests to the server processor, which, in turn, processes the requests and sends back replies. The requests and replies are communicated via an interconnecting network. This communication introduces overhead, which has an impact on the performance of a parallel application.

For a data-parallel application the programmer has two options for implementing the scheduling algorithm. The algorithm can either send a complete workunit to the requesting processor or merely an index to the workunit in the memory of the requesting processor. The former method is very straightforward but also more expensive. The latter method consumes less time but requires that the client processors have information about all workunits.

3.4. Discussion

It is a well-known fact that a scheduling algorithm plays a major role in achieving good speedups. Unfortunately, there is no single algorithm that is suitable for all parallel environments and applications; a number of scheduling schemes and algorithms have been developed and presented. In this chapter a concept of application-specific scheduling for data-parallel applications was introduced and studied with three scheduling algorithms. It was shown that providing additional information to a scheduling algorithm increases its capability to further improve the work balance among processors. It was concluded that an application-specific scheduling algorithm (based on the concept of a work pool) is able to produce very good scheduling results. The results are shown in detail in Chapter 5 and [Publication 4].

Chapter 4. MPIT

This chapter introduces the MPIT (MPI-Threads) programming paradigm that was implemented as part of the research work for this thesis. The MPIT is an efficient and user-friendly paradigm for writing data-parallel applications in networks of SMP workstations (SMP NOWs). The paradigm utilizes the best characteristics of shared and distributed memory environments, while hiding communication operations from the programmer. Also, the MPIT paradigm is capable of performing the automatic scheduling of workunits.

4.1. Introduction

The MPIT provides a new programming paradigm for SMP NOWs. The paradigm can be considered an extension of the MPI message passing library. The architecture of the MPIT is comprised of two parts. The first part is the MPI message passing library; the MPI is used to transfer data among workstations. The second part is the POSIX thread library; the thread library is responsible for performing the actual computation as well as the synchronization of the processors in a workstation. The MPIT programming paradigm has been implemented with C programming language and provides an MPI-like interface for the programmer.

The benefits of MPIT include:

1. Low resource utilization and overhead. The use of threads rather than processes is more efficient in an SMP workstation and introduces less overhead. For example, in a case where a context switch is required, it is more expensive to swap out a process than a thread. Furthermore, the MPIT allows for dynamic thread creation and termination. This provides the programmer with a tool to control the resource utilization of an application in a workstation.
2. Fast communication among processors in a workstation. Since only one process is created in a workstation and threads are used to utilize all processors, the communication occurs within the process. The communication within the process is substantially faster than communication among processes.
3. Low communication overhead through communication-computation overlap. The architecture of the MPIT allows for a fast communication start-up (completion time) that minimizes the impact of communication. This feature is achieved with a dedicated communication thread located in each workstation.
4. Support for heterogeneous networks of workstations. Each workstation can be configured to have any number of threads initially. Furthermore, the number of threads can be increased or decreased based on the load in a workstation or on the programmer's requirements for computing power.
5. Automatic scheduling functionality. The MPIT library has an implementation of the WorkPool scheduling algorithm. The scheduling functionality is hidden from the programmer and performed automatically by the communication thread. This

allows for fast scheduling that does not impact the execution of the actual application.

Since the MPIT is implemented on top of the MPI environment, it can support any programming model facilitated by the MPI. However, the MPIT is geared towards data-parallel computing. The user is allowed to define one function that is executed by all the worker threads after their creation. This function is where the user can control the execution of the worker threads. The user is not required to make all the worker threads run the same code segment. In fact, it is possible that each worker thread is assigned with a specific task turning the application into functional parallel software. The MPIT does not restrict the use of functional parallelism, although the scheduling mechanism is intended for the data-parallel programming paradigm; the automatic scheduling performed by the communication thread (see Subsection 4.7.) assumes that the data being distributed are separate workunits to be processed by individual threads in a data-parallel manner. Sometimes, an application combining the two programming models is the most suitable implementation.

The availability of threads instead of processes in a workstation slightly changes the programming model [Fah95]. The programmer has to consider issues such as synchronization and communication among threads in a workstation. For synchronization, the POSIX thread library offers a number of methods [Nor96]: mutual exclusion locks, condition variable, and semaphores. Since it is up to the programmer to implement the function executed by the threads in a workstation, he/she has to take care of the synchronization at the application level. For implementing communication the programmer has two options. The first option includes the direct utilization of the shared memory to facilitate the sending and receiving of messages between threads. This can occur through a memory read or write operation. A synchronization method is, possibly, required to guarantee that the variables used in the communication are not accessed simultaneously by more than one thread. The other solution for communication among threads is to use the MPIT send and receive operations. The communication thread is able to handle messages sent among threads in a workstation. This method provides automatic synchronization but is a slightly slower implementation.

The MPIT provides a thread-safe interface for the communication operations even if the underlying MPI implementation is not thread-safe. Thread-safety, with respect to the MPI, has been achieved by concentrating all MPI communication operations on one (communication) thread. Thus, thread-safety is guaranteed, if the worker threads utilize the communication routines provided by the MPIT library. If the worker threads make any calls to the standard MPI communication routines, the underlying implementation of the MPI library determines whether the operations are thread-safe.

4.2. Related Work

A number of articles have been published about using threads with MPI. A multi-threaded MPI implementation is discussed in [Pro01]. The authors discuss how to make an MPI implementation, *mpich*, thread-safe and multithreaded. The focus of the paper is on how to change the implementation of the MPI to support multi-threading rather than how to provide the programmer with a multi-threaded programming paradigm on top of MPI.

In [Chi98] the authors present a multithread communication library that is built on top of the MPI. The study considers a thread execution model where a master thread creates threads that

perform the computation. These threads may call only collective communication routines provided by the library. However, if data is to be received or sent, the threads initiate a communication call and enter a barrier. The communication routine is completed when all threads have initiated the communication call and the call has been completed. After the completion of the communication operation the threads are allowed to continue their execution. This kind of multithread communication library differs from the one presented in this thesis in various ways. The MPIT does not require that the threads in a workstation be executed according to a loosely synchronous model. However, the MPIT does not, currently, support collective communication routines. The work presented in [Chi98] does not consider scheduling, whereas the MPIT includes an application-specific scheduling algorithm.

The research presented in [Tan01] is a threaded MPI implementation for clusters of workstations called the TMPI (Threaded MPI). The TMPI implementation utilizes threads instead of processes in a workstation to take advantage of multiple processors. The communication among workstations is handled by a dedicated TMPI daemon thread in each workstation. The TMPI implementation offers a subset of the MPI commands found in the *mpich* implementation [Gro96]. The major differences between the TMPI and the MPIT are the structure of the TMPI and scheduling. The TMPI architecture consists of a number of layers similar to the architecture of the *mpich* implementation. However, the TMPI implements its own point-to-point and collective communication routines. The TMPI is not implemented on top of the MPI message passing library, whereas the MPIT paradigm is built on top of the MPI. The other difference between the TMPI and the MPIT is that the MPIT has a built-in scheduling mechanism that does not exist in the TMPI.

[Hai94] presents a system called Chant that resembles the MPIT environment. The Chant system provides point-to-point communication mechanisms for individual threads. It requires that communicating threads know that a communication operation is about to occur; the communication occurs in a synchronous fashion. In fact, the major difference in the Chant system compared to the MPIT paradigm is the way the communication is handled. The Chant system requires synchronous communication between two threads, whereas the MPIT paradigm allows for asynchronous as well as synchronous communication among workstations and threads.

In [Tan99] a programming paradigm, called COMPaS, for a cluster of SMP (Pentium Pro) workstations is presented. The paradigm utilizes a message passing mechanism called NICAM that was implemented by the authors. The NICAM is an active message –like system that supports remote memory operations with the use of message handlers [Par99]. The thread execution in a workstation is performed in a bulk synchronous fashion whereby a computation phase is followed by a communication phase [McC94]. Only a single thread (called the parent thread) is allowed to communicate with other workstations. The COMPaS system is a limited version of the MPIT environment in the sense that it does not support the free communication of threads and requires a bulk synchronous programming model. Furthermore, the NICAM message passing paradigm is not as portable as the MPI.

A programming paradigm called SIMPLE for clusters of SMP workstations is discussed in [Bad97]. The paradigm is a full programming environment for data-parallel applications. It provides a set of tools to write an application that is then distributed among workstations and threads in them. The necessary communication is implemented by the tools and handled by the system at runtime. Although the paradigm supports the MPI, the authors have developed

their own message passing library citing that the overhead introduced by the MPI is too excessive. The SIMPLE paradigm allows for threads to communication with each other and to be synchronized in a similar way than in the MPIT. However, the SIMPLE paradigm does not support heterogeneous networks of SMP workstations or facilitate scheduling as the MPIT does.

A runtime system called Nexus is presented in [Fos96]. The purpose of the system is to integrate threads and communication. This is achieved by the use of threads, global pointers, and remote service requests. Threads operate on global pointers by issuing remote service requests. The global pointer specifies the data to be processed and the target context that should process the data. With the use of a remote service request the data is communicated to the target context and processed. A separate thread on the target context is generated to perform the computation. All this takes place without interrupting the work of the target context. The context does not have to call a receive operation to retrieve the data and to process it. The Nexus system automatically handles all necessary operations on the target context. Although the system provides an interesting method (closely related to RPC without synchronization), it does not address issues such as scheduling.

In [Hen98] a programming paradigm implemented on top of the MPI message passing library is presented. This programming paradigm extends the BSP model to support communication. In addition, the paradigm defines a term called *virtual processes*. The virtual processes are the entities that run the code. The authors give examples about possible implementations of virtual processes. Threads are mentioned as one of the implementation options. The programming paradigm presented provides a limited functionality to the programmer. The computation is divided into four parts that include initialization, local computation, local result gathering, and global result gathering. Similar to the MPIT, it does allow threads to communicate with each other. However, the MPIT does not require that an application follow the BSP model.

Finally, the OpenMP programming model has been proposed as a solution to write parallel applications for shared memory environments [Büc02][Had02][Nik00]. Studies have also been conducted to incorporate the OpenMP with the MPI to provide a programming paradigm for distributed memory environments and networks of SMP workstations [Cap00][Hen00][Hu99]. The OpenMP programming paradigm provides a compile-time parallelization mechanism that allows for the programmer to gradually (one part at a time) turn a sequential application into a parallel application. The paradigm follows the fork-join model for thread creation and termination; threads are created for each time a parallel section in the code is encountered. What has made the OpenMP such a popular tool for writing application is the fact that the thread creation, maintenance, synchronization, communication, and termination are all hidden from the programmer. The MPIT programming paradigm presented in this thesis operates on a lower level than OpenMP, which gives more control to the programmer, but still hides certain functionalities from the programmer.

4.3. Architecture of MPIT

The MPIT is build on top of the MPI message passing library. The MPI provides the message transfer functionality for communication among workstations. Currently, the MPIT implementation offers an interface for the MPI point-to-point communication operations. However, due to the characteristics of the MPIT programming paradigm certain types of

collective communication routines are supported (such as a message can be sent to all threads in a workstation) as well as normal MPI collective communication routines can be used.

The implementation of thread functionality in a workstation is the heart of the MPIT programming paradigm. The internal components of the MPIT are depicted in Figure 4-1. The MPIT operates within a single process in each workstation. Worker threads are created inside the process according to a configuration parameter. The worker threads process data stored in the global (shared) memory. The global memory can also be used for communication among the threads. If the threads need to communicate with threads in other workstations, they have to use the services provided by a communication thread. A separate communication thread is created for each workstation. The communication thread is responsible for handling all incoming and outgoing messages.

Figure 4-1 also shows the message flow. When an MPI message is received from a network (Step 1), the communication thread stores it to the Data-In queue (Step 2) for immediate retrieval by a worker thread (Step 3). If a worker thread needs to send data, the data is stored to the Data-Out queue (Step 4). The communication thread calls an MPI send operation to initiate the message transfer (Step 5). The message is sent to the network by the MPI (step 6). With the help of the communication thread the MPIT programming paradigm achieves low communication-to-computation ratio increasing the performance of a parallel application.

The main functionality of the MPIT is hidden in the communication thread. The worker threads have an interface to the communication thread allowing them to send and receive messages to and from other workstations or threads (the interfaces are discussed later in this chapter). The MPIT send function allows for the worker thread to define a workstation and a thread in that workstation as a target of the communication operation. The worker thread can also use a special thread number to cause the message to be processed by any thread or all threads in the target workstation. The send operation creates a new entry in the Data-Out queue (Step 4). This is the main operation performed in the function called by the worker thread. Therefore, a communication request generates very little overhead and minimizes the communication-to-computation ratio (performance results are given later in this chapter). Also, a signal is sent to the communication thread as part of the function called by the worker thread. The signal indicates to the communication thread that new data has been entered to the Data-Out queue. The data is then sent by the communication thread to the receiving workstation. Another responsibility of the communication thread is to monitor the MPI input buffer for possible messages from other workstations. If a new message is discovered in the MPI message input buffer, the communication thread issue an MPI function call to retrieve the message. The retrieved message is stored in the Data-In queue. The Data-In queue holds all messages in order they were received. Once a worker thread issues a receive operation (Step 3), the Data-In queue is searched for a message that matches the message parameters (such as type and source). If no message is found, the worker thread either continues its execution or waits until a matching message is received. In the latter case, the worker thread relinquishes the processor on which it was running. This maximizes the utilization of processors. In the case where a message is found in the Data-In queue, the receive operation is very fast. It does not require access to the MPI buffer or memory copies; the data is returned from the Data-In queue that is located in the memory allocated for the process.

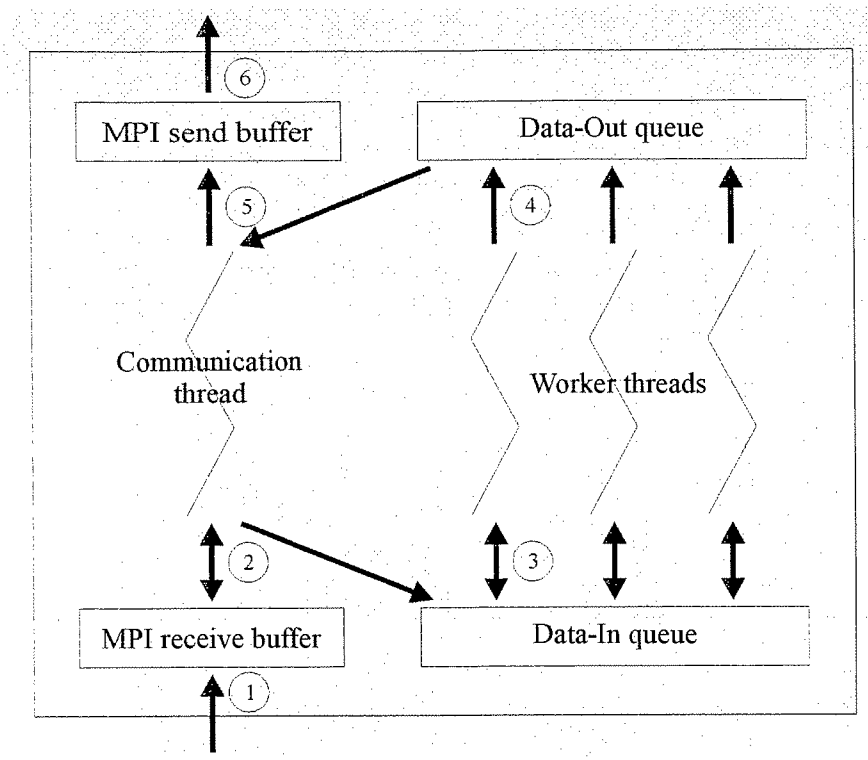


Figure 4-1. The internal components of the MPIT and the message flow. (1) An MPI message is received from a network. (2) The communication thread retrieves message and stores it to the Data-In queue. (3) Worker threads receive data from the Data-In queue. (4) Messages sent by the worker threads are stored in the Data-Out queue. (5) The communication thread initiates MPI send operations for the messages in the Data-Out queue. (6) An MPI message is sent out from the process.

The creation of worker threads in the MPIT is hidden from the user. Each thread is assigned a unique number within a workstation. This thread identification can be used by the programmer to identify threads and synchronize their execution. Furthermore, the programmer is capable of creating more threads as well as terminating existing threads during the execution of the application. This feature allows for the programmer to dynamically control the level of parallelism in a workstation.

4.4. Communication Thread

The communication thread is created during the initialization of the MPIT environment. The thread executes code that is included in the MPIT implementation. This code handles all communication operations within and between workstations. Thus, the MPIT programming paradigm facilitates threads sending messages to other threads in the same workstation and in other workstations. A message addressed to a thread in the same workstation never enters the network (not even the loopback interface). The message is processed by the communication thread in the sense that the message is moved from the Data-Out queue to the Data-In queue.

The implementation of the communication thread is relatively complicated, since it has to be able to perform operations initiated by the worker threads, incoming messages, and incoming scheduling requests (see subsection 4.7). Therefore, it is possible that the communication thread requires a dedicated processor. In such a case, the number of processors available in a workstation for the actual computation is decreased. In a dual processor workstation the dedication of one processor for the communication thread is not a feasible option. As a solution the MPIT paradigm implements a tightly coupled synchronization method that allows the communication thread to be active only when necessary. The synchronization scheme has been implemented with mechanisms provided by the pthread library. All (synchronization) signalling between threads and the MPI in the MPIT is handled with the condition variable primitive. Thus, it is important to note that the term *signal* refers to an internal signal within the pthread library that is used by the condition variable primitives. In general, the MPIT library does not impose any restrictions on how normal signals are handled on the worker threads. It is left up to the user to block the deliveries of unwanted signals. On the other hand, the communication thread only processes signals that cannot be blocked.

The synchronization mechanism in the communication thread is as follows:

- When a worker thread issues a send operation, the data to be sent is copied to the Data-Out queue. The function that performs the copy operation signals the communication thread that new data has been entered into the Data-Out queue. The signal wakes up the communication thread, which then performs the necessary communication operations.
- When a message arrives from another workstation it is first processed by the MPI. The MPIT can be tightly coupled with the MPI implementation. What this means is that after receiving a message from the network, the MPI signals the communication thread in the MPIT to process the message. Once the communication thread receives the signal, it issues an MPI receive command to retrieve the message and stores it to the Data-In queue. The tight coupling requires minor modifications to the MPI implementation. If such modifications cannot be made, the communication thread can be loosely coupled with the MPI. The loose coupling implements a polling mechanism where the communication thread checks for new messages at certain intervals.
- After the communication thread has retrieved a message and placed it to the Data-In queue, it signals all worker threads that are waiting to receive data. This is required since it is possible that a worker thread has issued a receive operation and the data has not yet been received by the communication thread. In this case, the execution of the calling worker thread is halted assuming that a blocking receive operation was used. Therefore, when the communication thread has retrieved and stored the received message, all threads waiting for messages are signalled. The signalled threads then try to match the new message with their requests. If no match is found, the threads go back to sleep to wait for another message from the network.

The execution of the communication thread has to be sequenced with the execution of the worker threads unless a separate processor has been dedicated for the communication thread or the number of worker threads is less than the total number of processors available in the system. Since applications have different requirements and preferences for communication and computation, the MPIT paradigm offers a mechanism for controlling the execution of communication and worker threads. This mechanism has been designed, but not yet

implemented. However, a short introduction of the MPIT thread priority mechanism follows. In the MPIT, the user is capable of assigning different priorities to the communication and workers threads. In fact, each worker thread can even have different priority. It should be noted that the priority scheme is based on the POSIX thread priorities. If the underlying thread implementation does not support thread priorities this option is not functional in the MPIT. With the proper priority scheme the user can give preference to one particular worker thread, or all of them, over the communication thread. In the latter case, the communication thread is only executed when no worker threads are executed. Thus, the worker threads are allowed to maximize their share of the processors at the expense of the communication thread. This kind of priority scheme leads to delays in the processing of messages sent by the worker threads and received from other workstations. However, if such a scheme is desired, assigning the worker threads with higher priorities than the worker thread boosts the parallel computation phase of an application. On the other hand, if the communication is a significant factor in the parallel computation, a higher priority can be assigned to the communication thread than to the worker threads. This scenario guarantees that the communication thread is executing whenever messages are to be processed and it will not be context switched before it decides to give up the processor. Another solution to improve the performance of the communication thread is, of course, to use a dedicated processor.

The main purpose of having a separate communication thread is to minimize the responsibility of the worker threads for communicating; by deploying the communication thread, the worker threads can focus on the actual computation. Thus, the communication thread provides the means for simultaneous communication and computation.

4.5. Worker Threads

The worker threads perform the actual computation. They are created during the initialization of the MPIT environment. The user needs to specify the number of threads created for each workstation and the function (with one argument) that the threads execute. The MPIT does not impose any restrictions or limitation on what the worker threads can do; the worker threads are not restricted from calling any library calls. The programmer has to consider the issues involving thread-safety among the worker threads, since the MPIT is not capable of providing thread-safety for threads within a workstation (as noted earlier, the MPIT does provide thread-safe MPIT communication routines). Furthermore, it is up to the programmer to handle communication within the threads in a workstation and to perform any required synchronization. The programmer can use the MPIT send and receive operations to handle communication among threads (see Subsection 4.6.2). The MPIT also provides a set of functions to perform thread synchronization. At the end of the computation, the programmer must make sure that the execution of the worker threads is properly terminated. The MPIT environment cannot be terminated until all worker threads have completed their work or have been cancelled.

4.6. Programming Interface

The programming interface is divided into three parts. The first part consists of a set of function calls to set up and terminate the MPIT environment. The second part includes functions to perform point-to-point communication among workstations. The third part is comprised of function calls for thread synchronization and maintenance.

4.6.1. Setup and Termination of MPIT

The MPIT environment must be initialized in a manner similar to the MPI. It is important to notice that the MPI environment needs to be initialized prior to the initialization of the MPIT environment. The initialization of the MPI environment was not included in the MPIT initialization process, since that allows for an application to be executed as a single threaded MPI application (a conventional MPI application) before the MPIT environment is initialized. Furthermore, for testing purposes in a stand-alone (SMP) workstation, this approach makes it possible to initialize the MPIT environment without initializing the MPI environment.

The MPIT initialization function is shown in Table 4-1. The first argument of the function is the name of a file that contains information about the workstations in the network. The file specifies the names of the workstations, the number of threads to be created in each workstation, and the relative powers of the workstations. The relative power information can be used by scheduling algorithms to optimize the distribution of workunits with respect to the computing capabilities of the workstations.

The initialization of the MPIT environment creates the communication thread and the worker threads. The creation of the communication thread includes setting up of the Data-In and Data-Out queues, and initializing the signalling mechanisms. Once the communication thread is successfully created and initialized, the worker threads are created. These threads start to execute the function defined in the *MPIT_Init* function call (with the argument defined also in the *MPIT_Init* call). Once the *MPIT_Init* function call returns, the communication thread is ready to receive and send messages, and the worker threads execute their code. The main thread that called the *MPIT_Init* function has two options. (1) It can call the MPIT termination function and relinquish the processor until all the worker threads have terminated. (2) It can participate in the computation as any another worker thread.

The initialization of the MPIT environment takes approximately 1.1 milliseconds in a Pentium III 750MHz workstation running the Linux operating system. The corresponding times for TCP (connection establishment) and MPI are 1.0 and 3.0 milliseconds, respectively. Thus, the initialization of the MPIT environment is very close to the initialization time of a TCP socket connection.

Table 4-1. MPIT initialization and termination function calls.

MPIT function call	Description
`MPIT_Init(machineInfoFile, threadFunc, threadFuncArgs)`	Initialize the MPIT environment.
`MPIT_Terminate()`	Terminate the MPIT environment.

The MPIT termination call is also shown in Table 4-1. This function should only be called by the main thread. The function first waits until the communication thread and all the worker threads are terminated. Then, the termination function releases the memory allocated for the communication thread, and the Data-In and Data-Out queues. The function does not terminate the MPI environment. Therefore, the execution of the application can continue after the termination of the MPIT environment as a normal MPI application. Furthermore, it is possible to reinitialize MPIT without restarting the application.

4.6.2. Point-to-Point Communication

The MPIT programming paradigm offers a set of point-to-point communication operations to carry out data transfers from one workstation/thread to another. The paradigm does not impose any restrictions on when a thread can send or receive data; thus, all threads are allowed to send and receive data as required. Due to the implementation of the MPIT, data transfers are handled by the communication thread. Therefore, the communication routines available for sending messages only add the messages to the Data-Out queue allowing the thread to continue its execution immediately.

In order to identify the sender and receiver of a message, the following naming scheme was implemented. Each workstation has a unique process identification (rank) that is assigned when the MPI environment is initialized. This identification is used by the worker threads to send out messages to other workstations. Each worker thread has an identification number that uniquely distinguishes it from other threads in a workstation. The identifier is assigned to the worker thread when it is created. The identifier 0 is reserved for the main thread, so the worker threads created have identifiers starting from 1. This kind of naming scheme requires that in order to uniquely identify a thread in the MPIT, workstation and thread identifiers must be known.

Table 4-2. MPIT point-to-point communication calls.

MPIT function call	Description

`MPIT_Send(`
`  data, numElmnts, type, dstProc,`
`  dstThrd, tag, comm`
`)`
	Send a message to another workstation.

`MPIT_Recv(`
`  data, numElmnts, type, srcProc,`
`  srcThrd, tag, comm, mpiSts`
`)`
	Receive a message from another workstation (blocking).

`MPIT_Irecv(`
`  data, numElmnts, type, srcProc,`
`  srcThrd, tag, comm, mpiSts`
`)`
	Receive a message from another workstation (non-blocking).

`MPIT_RecvAny(`
`  data, numElmnts, type, srcProc,`
`  srcThrd, tag, comm, mpiSts`
`)`
	Receive any message from any workstation (blocking).

`MPIT_IrecvAny(`
`  data, numElmnts, type, srcProc,`
`  srcThrd, tag, comm, mpiSts`
`)`
	Receive any message from any workstation (non-blocking).

Table 4-2 shows the point-to-point communication calls currently available in the MPIT. As seen in the table, the function arguments are very similar to the MPI communication operation arguments. The main difference is the additional argument srcThrd/dstThrd that defines the source/target thread. It should be noted that there is only one version of the send operation. Since the communication is handled by the communication thread, a blocking send operation is not supported (although the naming convention implies that it is a blocking operation). Furthermore, a worker thread can send a message to a workstation without

specifying a particular thread that must receive the message. This allows for flexibility, since any thread capable of handling the message can do so. It is also possible to specify a thread identifier that causes all threads in the receiving workstation to process the message. Both of the normal receiving operations work similarly to the MPI functions. In addition, the MPIT paradigm offers two new function calls to receive data. The *receive any* function calls allow a thread to receive any type of data from any workstation; the size of the data is not predetermined and neither is the tag. These functions set values for all arguments, except the communicator, to reflect the data received.

4.6.3. Thread Synchronization and Maintenance

The MPIT paradigm handles synchronization within the MPIT function calls; all the function calls to the MPIT library are thread-safe. The MPIT keeps track of the worker threads and communication thread. The threads are automatically created during the initialization of the MPIT environment and joined at the termination of the MPIT environment.

However, during the parallel computation phase the programmer is responsible for providing synchronization methods to control the execution of the threads. Table 4-3 shows the thread synchronization calls provided by the MPIT paradigm. The *MPIT_ThrdBar* function implements a thread barrier within a workstation. The *MPIT_ThrdSusp* and *MPIT_ThrdRes* function calls are used to suspend and resume the execution of a thread. The suspend function call blocks the execution of the calling thread until another thread uses the resume function call to allow the suspended thread to continue its execution. All these function calls do not limit the programmer from implementing other synchronization methods provided by the POSIX thread library [But97].

In addition to the synchronization operations the programmer is provided with a set of function calls to dynamically control the number of threads in a workstation. These function calls are shown in Table 4-4. The *MPIT_ThrdAdd* function creates a new thread that starts executing the code specified in the function arguments. The *MPIT_ThrdTerm* function call terminates a thread with the help of the cancellation mechanism provided by the pthread library. These functions become useful in environments where threads are generated for a particular task and are terminated once the task is completed, or when the programmer wants to control the resource utilization in a workstation. The programmer should not call the *MPIT_Init* function to create new threads. In order to maintain a coherent state of the MPIT in a workstation the programmer should also refrain from creating new threads with the *pthread_create* function. This function does not properly register the new thread with the MPIT environment causing problems when the environment is terminated. The same reasons apply for having the thread termination function call.

Table 4-3. The thread synchronization function calls in the MPIT paradigm.

MPIT function call	Description
`MPIT_ThrdBar()`	Thread barrier.
`MPIT_ThrdSusp()`	Suspend the execution of the calling thread.
`MPIT_ThrdRes(thrdId)`	Resume the execution of the specific thread.

Table 4-4. The thread maintenance function calls in the MPIT paradigm.

MPIT function call	Description
`MPIT_ThrdAdd( threadFunc, threadFuncArgs )`	Add a new thread.
`MPIT_ThrdTerm(threadId)`	Terminate a thread.

4.7. Scheduling in MPIT

The MPIT includes an implementation of the application-specific WorkPool algorithm. In order to use the internal scheduling mechanism one workstation has to be defined as the master workstation. The responsibility of the master workstation is to respond to requests for work from threads in workstations (including the master itself) participating in the computation. These requests are sent by the threads that are out of work. The scheduling scheme allows for a single thread to issue a request for more than one workunit at a time. Thus, the thread is able to retrieve workunits for other threads in the same workstation.

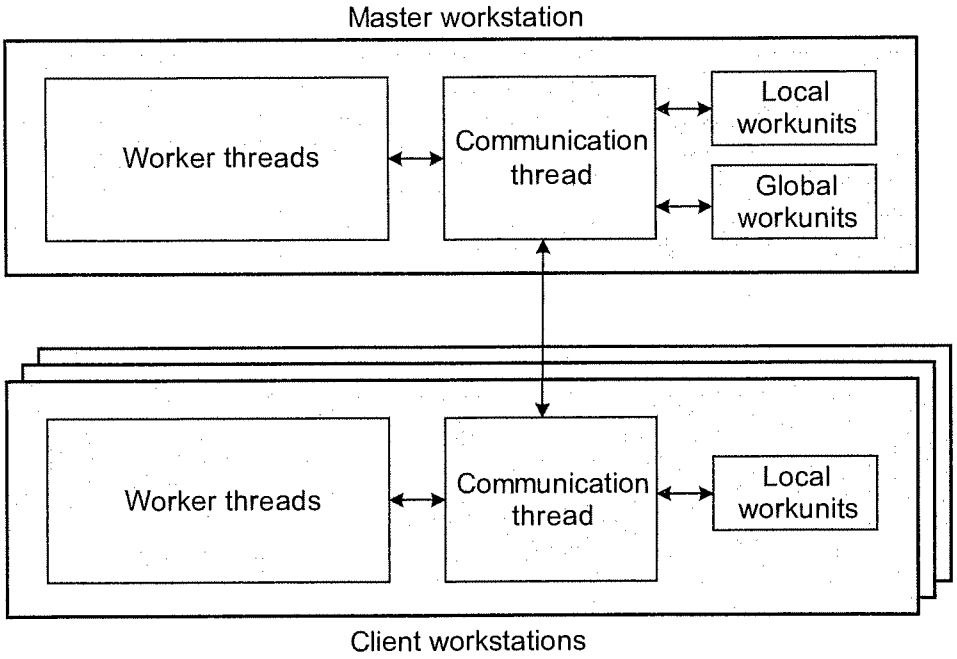


Figure 4-2. The scheduling mechanism in the MPIT.

Figure 4-2 illustrates the scheduling mechanism in the MPIT. The master workstation has knowledge of all (global) workunits. These workunits are scheduled among workstations (and the threads within them) based on requests received by the master workstation. If a thread is requesting more than one workunit and the master can fulfill the request, one of the workunits is given to the calling thread and the others are stored locally (in the local workunits buffer) in the workstation. When the next thread requests a workunit in the same workstation, the workunit is retrieved from the local workunits buffer rather than sending a request to the master workstation. If no workunits exist in the local workunit buffer, a request is generated

and forwarded to the master workstation. With the help of the local workunit buffer the number of requests that have to be sent over the network to the master processor is reduced. However, in order to take advantage of this scheme the threads much request more than one workunit at a time.

The scheduling is implemented as part of the communication thread. The scheduling messages (requests) are separated from the conventional data messages by the communication threads. These messages are sent to the scheduler (internal component of the communication thread) for processing. The part of the communication thread that performs the scheduling is denoted as the local scheduler. In the case of the master workstation it is called the master scheduler. As Figure 4-4 illustrates the schedulers handle the generation of requests, the transmission of requests, the request processing in the master workstation, and storing of workunits to local workunit buffers. Since the scheduler is implemented in the communication thread, the communication required to retrieve more work overlaps with the computation. The centralized processing of requests by the scheduler in a workstation allows for more control over the requests. For example, in a workstation the local scheduler prevents two or more requests from being active at a time; if the first request retrieves an adequate number of workunits, the second request is redundant. It is important to realize that the master workstation has a local workunit buffer as well. In fact, threads in the master workstation can send requests for work, and the scheduling mechanism works the same way as for any other workstation. The only difference is that no messages have to be sent over a network to request and retrieve the workunits.

The scheduling functions are shown in Table 4-5. To initialize the scheduling, all workstations have to call the *MPIT_SchdMaster* function to define the master workstation for scheduling. This function call is required in order to initialize the local scheduling client and to let it know to which workstation to send the scheduling requests. The *MPIT_SchdInit* function is only called in the master workstation; it initializes the master scheduler by proving information about the workunits, their computational requirements (weights), and their number. The information provided is immediately processed; the workunits are sorted in descending order according to their computation requirements starting with the most time-consuming workunit. The *MPIT_SchdActv* and *MPIT_SchdTerm* functions turn the automatic scheduling on and off in the master workstation. These functions allow for the programmer to control when the scheduling occurs. For example, scheduling should be disabled while workunits are being added or removed. The last two functions, *MPIT_SchdGet* and *MPIT_SchdIget*, are used by the threads to request more work. As mentioned earlier, a thread can request more than one workunit at a time. However, only one workunit is returned each time the *MPIT_SchdGet* function is called. Subsequent calls to the *MPIT_SchdGet* return a workunit from the local workunit buffer until its empty. A request to the master workstation is generated and sent, if the local workunit buffer becomes empty. The master workstation sends a special, end-of-data message to the requesting workstation, if all workunits have been distributed. This message is then returned for each subsequent call to the *MPIT_SchdGet* function by the local scheduler to indicate to the calling threads that there are no more workunits available. The *MPIT_SchdIget* function operates similar to the *MPIT_SchdGet* function except it does not block the execution of the calling thread. It should be noted that the blocking occurs only when a request is sent to the master workstation. Therefore, the *MPIT_SchdIget* function can be used to pre-fetch workunits assuming that the local workunit buffer is empty. However, if workunits exist in the local workunit buffer, they can be retrieved with either of the two functions. The local scheduler takes care of all necessary

synchronization required to prevent more than one request for work from being active at a time, as well as the synchronization required to make a decision from where to retrieve a workunit.

Table 4-5. The function calls for scheduling provided by the MPIT.

MPIT function call	Description
`MPIT_SchdMaster(procId)`	Specify the master for the scheduling.
`MPIT_SchdInit( workunits, weights, numWorkunits )`	Initialize scheduling (workunits) with MPIT.
`MPIT_SchdActv()`	Enable scheduling.
`MPIT_SchdTerm()`	Disable scheduling.
`MPIT_SchdGet( workunit, numWorkunits )`	Retrieve the specified number of workunits.
`MPIT_SchdIget( workunit, numWorkunits )`	Retrieve the specified number of workunits (non-blocking)

The automatic scheduling performed by the communication thread has several benefits. First, worker threads are not affected by the scheduling; they can continue their execution uninterrupted on both sides (master and client). Thus, the MPIT allows computation and scheduling to overlap in addition to computation-communication overlap. Second, the implementation of the scheduling algorithm is in the communication thread; the user does not have to consider implementing a scheduling algorithm. The communication thread automatically performs the scheduling when it has been activated. Third, the function calls to request a workunit from the master workstation actually facilitate the retrieval of multiple workunits per request. What this means is that a thread, rather than requesting work just for itself, can request work for all threads in the workstation. This allows for predictive scheduling whilst minimizing the number of scheduling requests. This reduces the number of communication operations as well as the work done by the communication threads. Finally, the non-blocking get function provides a mechanism to pre-fetch workunits while current workunits are still being processed. With this mechanism the threads finishing their work can immediately retrieve a new workunit from the local workunit buffer rather than sending a request to the master workstation and waiting for a reply.

4.8. Performance Results

The MPIT paradigm has a number of theoretical benefits, which have already been discussed in this chapter. More practical MPIT results are given in this subsection to show the real performance of the MPIT paradigm as well as to validate the theoretical benefits. For testing purposes, an application was written that performs calculations in a data-parallel fashion. In the MPIT environment the workload is distributed equally among the workstations. The workunits assigned to a workstation are processed by the threads (one workunit at a time) in the workstations. The corresponding MPI implementation distributes the workload among

processes in all workstations. The number of processes in a workstation is determined by the number of available processors. Each process handles the workunits assigned to it without any interactions between other processes in the same or any other workstation. Thus, the application does not utilize the scheduling mechanism provided as part of the MPIT library.

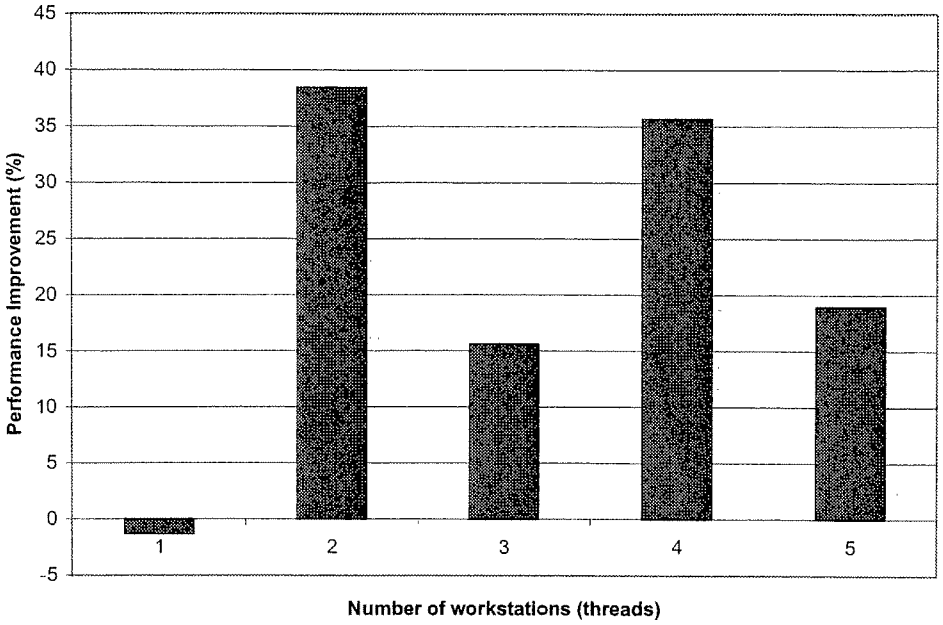


Figure 4-3. The performance of an MPIT-based simulation.

The results from the two versions of the simulator are shown in Figure 4-3 and [Publication 5]. The figure shows the results from a test run in a single 4-processor Linux workstation and a dual processor Linux workstation. The results for 1 to 4 threads were obtained when the application was run in the quad processor workstation. The result with 5 threads was achieved by utilizing the 4-processor workstation and one processor from a dual Linux workstation. At best the MPIT is over 35 % better than the MPI. This is can be attributed to the use of the shared memory which allows all threads to work on all workunits. The fluctuation in the results is explained by the uneven initial distribution of work among the threads. In addition, the results for the test where one thread is deployed shows the overhead introduced by the MPIT. The overhead is approximately 1.3 %.

4.9. Discussion

There are MPI implementations that support communication via a shared memory, if the communicating processes are located in the same workstation. However, the implementations still create more than one process in a workstation, whereas in the MPIT one process includes all the entities (threads) that execute the code. The threads share the memory allocation for the process. Thus, communication is faster through this memory than inter-process communication through a pipe, for instance. The utilization of the shared memory comes with issues that the programmer should be aware of, such as memory protection, and the synchronization of threads. These issues are left for the programmer to consider and

implement as necessary. In addition, the MPIT does not provide a thread-safe implementation of the underlying MPI communication library. Thus, the worker threads should refrain from calling any MPI routines, and utilize only the MPIT communication routines provided.

The MPIT library allows the receiving side of a message transmission to retrieve any number and type of data. What this means is that the receiver thread does not have to know how many data elements it should receive when it makes a call to the receive function. The MPIT library returns the data to the receiver and sets function arguments based on the message received. Furthermore, the number of broadcast messages in the MPIT is reduced, since only one message per workstation is required. This is because in the MPIT a message can be sent to a workstation that is eventually received by all threads in the workstation.

The automatic load balancing is a feature not found in any other proposed systems that are even vaguely similar to the MPIT. The application-specific WorkPool load balancing algorithm distributes workunits among workstations and threads without load migration. The scheduling is handled by the communication threads that reside in each workstation without interfering in the execution of the worker threads. The worker threads initiate a scheduling operation with a request for more work. The communication thread on the requesting workstation formulates a special message that is sent to the master workstation for processing. The master workstation sends a response to the workstation that contains the requested workunits or a special message indicating that all workunits have been processed. The MPIT library also offers a mechanism to pre-fetch workunits and to retrieve more than one workunit at a time. Although the scheduling mechanism is hidden from the programmer, it is the programmer's responsibility to determine the scheduling parameters for each workstation/thread; this involves considering whether to use the standard mechanism or pre-fetch workunits and how many workunits to request at a time.

Due to the fact that the MPIT programming paradigm is implemented with the MPI and the POSIX threads, the portability of the paradigm is optimized. Since the MPI and the POSIX threads have achieved substantial support from software vendors, the portability of the MPIT programming paradigm should be very good. Furthermore, the MPIT does not introduce any operating system or hardware specific requirements. In theory, the MPIT library works with any MPI implementation assuming proper MPI interface functions are exposed to the MPIT. It should be noted that in order to implement the tightly coupled communication mechanism between the MPI and the communication thread, the MPI implementation must be modified. Therefore, if the tightly coupled interface is required, the MPI implementation distributed with the MPIT implementation should be deployed. With the loose coupling, any MPI implementation can be used as long as it fulfills the MPIT interface requirements.

The MPIT performs all necessary operations to guarantee thread-safety for the communication operations provided as part of the MPIT. Thus, even a non-thread-safe version of MPI can be deployed with the MPIT. However, the MPIT does not enforce the thread-safe execution of the worker threads. The programmer is required to either use thread-safe library calls or implement synchronization to provide thread-safety among the worker threads.

Chapter 5. Case Studies of Mobile Network Simulators

Mobile network simulation has been a popular area of research due to the substantial increase in mobile phone users; the past decade has seen an unprecedented number of new subscribers. With the help of simulation, mobile equipment manufactures and network providers are able to test their base stations, mobile phones, and networks before investing large sums of money into this infrastructure [Hei96].

This chapter introduces two sequential mobile network simulators that were used as the case studies for this thesis. The simulators were implemented with C and C++ programming languages. Neither of the simulators was originally designed to run in parallel in any kind of multiprocessor system. The sequential simulators were implemented at Nokia Research Center in Helsinki, Finland.

The parallel implementations of the simulators utilize the knowledge gained from the parallel environments, and the algorithms studied and developed in this thesis. Slightly different implementations were developed for each environment [Hut98][Hut00]. In addition, all three application-specific scheduling algorithms were tested in each environment with one of the simulators. Prior to presenting the results, the two simulators and their parallel implementations are discussed.

5.1. GSM Network Simulator

The GSM network simulator is used to calculate a coverage area of a base station. The purpose of the simulator is to act as a tool for mobile network providers; it helps them in finding optimal locations for base stations in urban environments. The simulator calculates field strengths generated by a base station over a map given as an input parameter. Based on the location of the base station, the field strengths are estimated in accordance with radio propagation laws [Feu94][Rap96]. The field strength describes the reception and transmission quality of a mobile station in different locations of the map. It is important to find an optimal location for a base station in order to provide a signal of good quality to all users and to maximize frequency reuse. This leads to a minimal number of dropped calls and a longer battery life for mobile phones. Figure 5-1 depicts a sample map of Helsinki used in the simulations. The coverage calculated is indicated by the shaded areas. Although, the results are given for receiving points that are inside buildings, indoor ray propagation was not implemented. Therefore, the results shown for indoor receiving points are not accurate. It should also be noted that the only obstacles considered in the simulator were buildings.

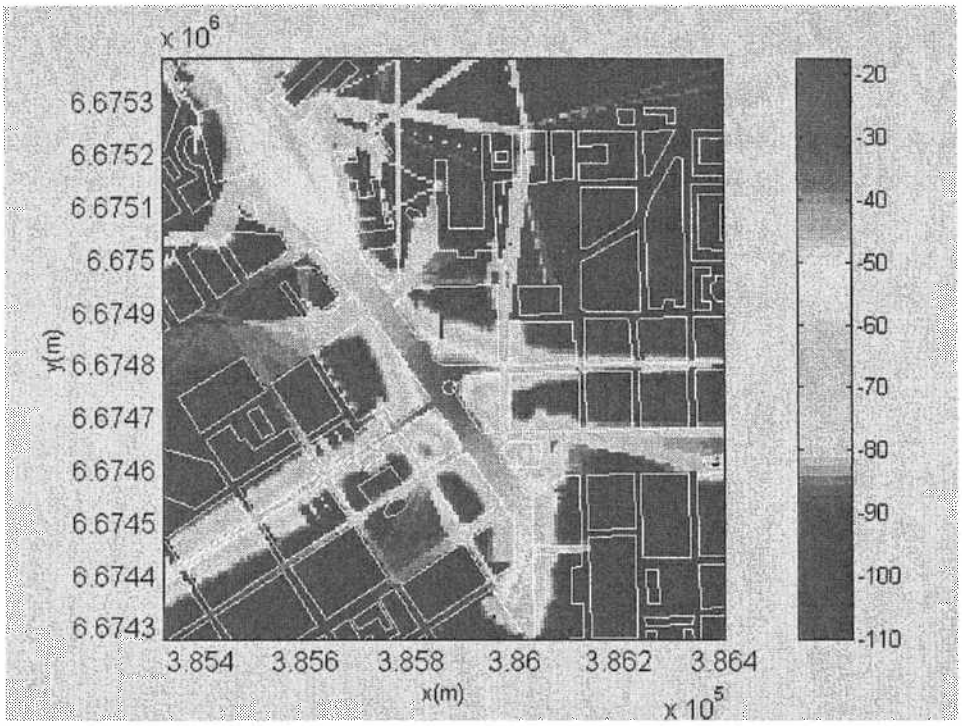


Figure 5-1. The calculation area and results generated by the GSM network simulator. The small circle in the middle of the figure illustrates the location of the base station.

Prior to the simulation, a map is divided into receiving points that hold the field strength information at the end of the simulation. Figure 5-2 shows an example of the receiving point generation. The receiving points are always square shaped and their size is determined by the accuracy requirements of the results. Since a single receiving point represents the average field strength over the area it covers, the size of the receiving point has a substantial impact on the accuracy of the coverage calculated. Furthermore, the size of a receiving point affects the total execution time of the simulation. The execution time is inversely exponential to the size of the receiving points; if the size of the receiving points is decreased by 50%, the execution time increases 200%.

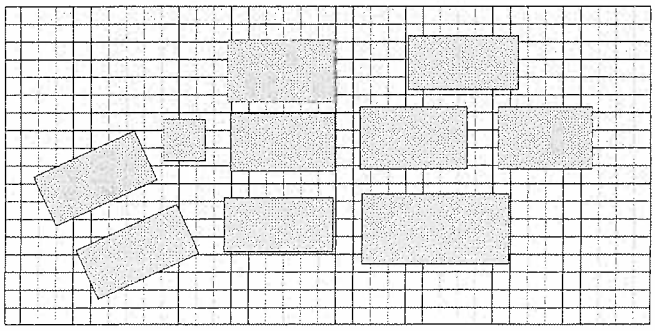


Figure 5-2. The generation of receiving points .

5.1.1. Simulation

Although the map shown in Figure 5-2 is two dimensional, the simulation is carried out in all three dimensions. The simulation is comprised of two phases. First, the ray propagation over the roofs of the buildings is computed. Second, the simulation of ray propagation on a horizontal level in the street canyons is performed. Figures 5-3 and 5-4 illustrate the two simulation phases.

The first phase, vertical ray propagation, includes measuring a distance between a receiving point and a base station, observing the obstacles between the two points, and calculating the field strength for the receiving point based on the laws of physics [Rap96][Sip96]. This procedure is repeated for each receiving point. Due to its low computational requirements the first phase of the simulation accounts for only 10% of the total execution time. Nevertheless, its contribution is large enough to justify a parallel implementation. The second phase, horizontal ray propagation, of the simulator utilizes ray tracing. However, the ray tracing method deployed in the simulator is not the standard method. In standard ray tracing, rays are launched either from a base or mobile station and are traced until they reach their destinations or their powers drop below a set threshold. The simulation of such a model is generally very time-consuming. Thus, the model was improved by introducing the concept of line-of-sight (LOS) polygons, which reduce the number of rays traced substantially. Instead of tracing a single ray, the simulator creates a LOS polygon for each corner of obstacles (buildings). Figure 5-5 illustrates a LOS polygon. The area covered by a LOS polygon is the area in which a ray diffracts from the corner where the polygon was created. A diffraction occurs when a ray arrives at a corner of a building and disperses in all possible directions, creating a large number of new rays. Therefore, the receiving points inside the LOS polygon are where the newly created rays eventually arrive, and the field strengths of which are updated. In order to update the field strength values of the receiving points inside a LOS polygon, the strength of the diffracted signal has to be known. Prior to processing the LOS polygons, the initial signal strengths of the rays at each corner are computed. For this procedure a traditional ray tracing method is used.

The coverage calculated by the simulator compared to real measurements has shown that the simulator is capable of producing very accurate estimations concerning the coverage of a base station. Therefore, the simulator does not consider ray propagation through reflections, which further reduces the complexity of the simulator.

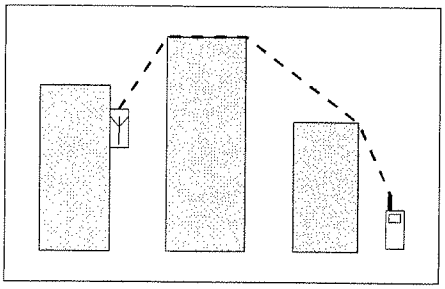


Figure 5-3. The vertical ray propagation example.

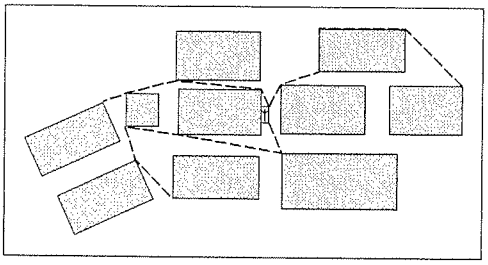


Figure 5-4. The horizontal ray propagation example.

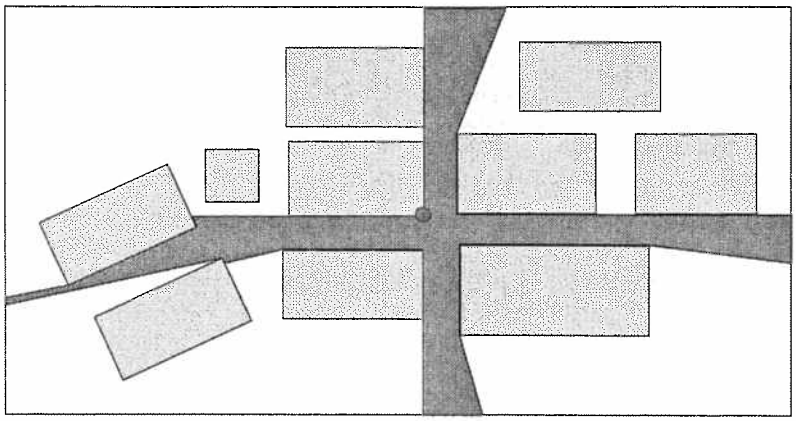


Figure 5-5. An example of a line-of-sight (LOS) polygon. The LOS polygon is the darker shaded area. The corner to which the LOS polygon was generated is depicted with a large black dot in the middle of the figure.

The first performance tests for the sequential implementation were performed immediately after the validations tests were completed. Table 5-1 shows the simulation times with different receiving point sizes in a Pentium 750 MHz Linux workstation. An adequate accuracy for the results was achieved with 4 x 4 meter receiving points. However, for commercial use the receiving point size cannot be larger than 2.5 x 2.5 meters. As the results show the execution times increase significantly when the size of the receiving points is decreased. The execution times for the most accurate simulations were far too excessive to be considered suitable for interactive usage. As a result, the sequential simulator code was reviewed and further optimized in order to boost performance. Unfortunately, the optimization process did not manage to produce the desired level of performance increase. Therefore, parallel processing was considered as a solution to decrease the execution times without compromising the accuracy of the results. The subsequent sections discuss the parallelization process in general, and then focus on the implementations of the simulator to a shared memory environment, a distributed memory environment, and a network of workstations.

Table 5-1. The execution times of the sequential GSM network simulator.

Grid size (m)	Execution time (sec)
10 x 10	27.97
4 x 4	168.80
2.5 x 2.5	433.72

5.1.2. Parallelization

Regardless of the fact that the simulator was not designed for parallel execution, the structure of the simulator was well-suited for data-parallel computation. Each individual LOS polygon could be executed in parallel, since no dependencies between the polygons existed. However, it was possible that two or more LOS polygons overlapped causing more than one processor to modify the same receiving points simultaneously. This feature called for a synchronization method to guarantee the correctness of the results in shared memory environments. In the following subsections the implementation and impact of the synchronization method are discussed.

The first implementation of the GSM network simulator included all three scheduling algorithms. However, it was concluded based on the results (see Subsection 5.1.3) that the application-specific WorkPool algorithm (Algorithm 3) was superior to the other two algorithms in each parallel environment. Due to the fact that Algorithm 3 showed the best performance, only the implementation of Algorithm 3 is explored.

The first phase of the simulation, vertical ray propagation, contributes approximately 10% to the total execution. As a result of its minor impact, the implementation of the application-specific WorkPool algorithm was not justified. A more simple scheduling algorithm was suitable, since the workunits which are the receiving points, all contained an equal amount of work. Therefore, a static load balancing algorithm was developed. The algorithm distributes the workunits equally among the processors. On the other hand, the application-specific WorkPool algorithm was implemented in the second phase of simulation, horizontal ray propagation. The workload consisted of the LOS polygons, the sizes of which varied drastically. The environment-specific implementations are discussed in the next two subsections.

Shared Memory Environment

In the shared memory environment, the sequential GSM network simulator was amended to create and maintain threads. The threads were used to process a number of LOS polygons simultaneously on multiple processors. A synchronization method (mutual exclusion lock) was implemented to protect the receiving points from being updated by more than one thread at a time. A mutual exclusion lock was associated with each receiving point to minimize the impact of the synchronization procedure on the execution of other threads.

The implementation of the scheduling algorithm required a global variable that held an index to the next unprocessed LOS polygon. Once a thread was ready to process a workunit, it obtained the current value of the index and increased the value by one. However, since it was possible for more than one thread to access the global variable (index) at a time, a synchronization method was put into place. Prior to fetching the index and increasing its value, a thread was required to acquire a mutex lock. After updating the value of the global variable the thread released the mutex lock and proceeded with processing the new workunit pointed to by the index. In order to utilize the advantage of the application-specific algorithm, the returned index always pointed to the most computationally complex workunit. For this to happen the workunits were sorted in descending order by the amount of computation they required. Therefore, the total overhead introduced by scheduling in shared memory environments was incurred by acquiring and releasing the mutex lock and determining the processing order of the workunits.

Distributed Memory Environment and Networks of Workstations

Due to the differences in memory architectures, the implementation of the data-parallel version of the simulator for distributed memory environments and NOWs was somewhat different. For example, there was no need for a synchronization method to protect the receiving points, since each processor worked with its own copy of the receiving points. The receiving points were gathered to one processor that produced the final results at the end of the computation. The gathering required communication over the interconnecting network, which contributed to the overhead in the distributed memory environments and NOWs.

However, this communication did not impact the actual parallel computation phase, since the only communication required during the computation phase was the transmission of scheduling messages.

For the scheduling, one processor was assigned to act as a master. Its responsibility was to maintain and distribute the workunits among the other processors. Since the application-specific WorkPool algorithm was implemented in the distributed memory environments and NOWs, the master processor was responsible for the following operations in the course of the simulation:

- Sorting of workunits according to their computational requirements.
- Reception of workunit requests from processors.
- Fetching and updating the index pointing to computationally the most complex unprocessed workunit.
- Sending of replies to the requesting processors.

The master processor either sent an index to the workunit the processor was supposed to process next or to the actual workunit. The algorithm was implemented so that it supported both methods. The request-reply procedure required communication between the master processor and the requesting worker processor. As was the case with sending the final results to the master processor, the communication was the most substantial part of the overhead generated by the scheduling algorithm. Another reason for the overhead was the queue of requests on the master processor. If more than one worker processor was requesting work, the requests were handled sequentially causing delays in the worker processors. Therefore, the communication was not the only delay experienced by the worker processors when waiting for a reply from the master processor. A pre-fetching mechanism could have been implemented to anticipate the need for more work and minimize the impact of the delay. For example, an implementation of the simulator in the MPIT environment could have taken advantage of the pre-fetching functionality in the scheduling mechanism provided.

5.1.3. Results

In this subsection, the results from the three environments are shown. The performance results are indicated as speedups, since the environments are not comparable from a hardware point of view; the execution times varied significantly from one environment to another.

Figure 5-6 illustrates the speedups achieved with the GSM network simulator in the three environments. Note that the speedups are shown for the complete application, and not simply for the parallel computation phase. Differences in the number of processors in the environments are explained by the availability of processors. The shared memory environment (Compaq AlphaServer8400) had only 8 processors, whereas in the distributed memory environment (Cray T3E) and NOW (PC cluster) 32 processors were available. In fact, the NOW was comprised of 16 dual processor workstations.

The results show that the shared memory environment is not capable of performing as well as the distributed memory environment and the NOW. The shared memory environment suffers from the thread synchronization required to guarantee sequential access to receiving points. However, the best performance was achieved when the number of threads exceeded the number of available processors. This indicates that the threads perform time-consuming

operations that cause the operating system to relinquish the processor in favour of another thread. The distributed memory environment and the NOW performed quite similarly. The distributed memory environment (Cray T3E) was a dedicated system, which explains the stability of the results. On the other hand, the NOW suffered from occasional load fluctuations generated by other users. Overall, the results were heavily impacted by the serial part of the simulator. The significance of the sequential code became more obvious when more processors were deployed. This, in addition to the overhead introduced by the parallel computation, caused the results be sub-linear. However, it should be noted that the speedups for the parallel computation phases exhibited near linear speedups even with a large number of processors.

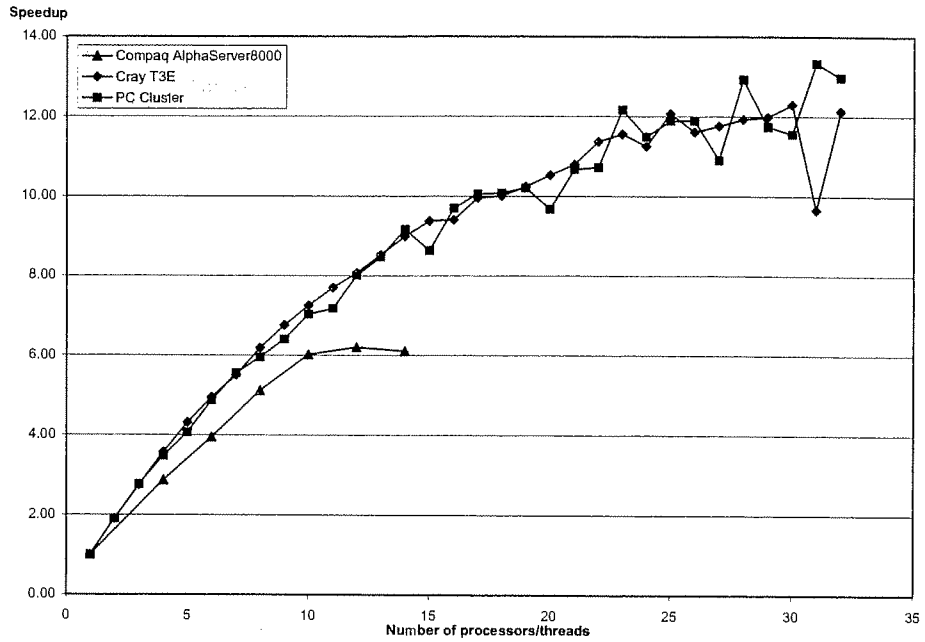


Figure 5-6. Speedups achieved in the three memory environments with the data-parallel version of the GSM network simulator.

Scheduling Algorithms

All three scheduling algorithms were implemented and tested with the GSM network simulator. Here, the results are shown with respect to the amount of work processed by each processor when the simulator is run in a 4-processor Linux workstation. It should be noted that the amount of work, which is used to illustrate the distributed workunits, is directly proportional to the processing time of that work.

Detailed results for Algorithm 1 are shown in Table 5-2. The table indicates work balance results when both versions of Algorithm 1 are deployed. It is obvious that the application-specific algorithm is capable of producing a more optimal load balance. Both the standard deviation and the co-efficient of variation imply that the application-specific algorithm is over an order of magnitude better than the generic algorithm.

Table 5-2. A comparison of the generic scheduling algorithm 1 and the application-specific scheduling algorithm 1. The figures specify the amount of work (receiving points) processed by each processor

Processor Id	Generic Algorithm 1	Application-Specific Algorithm 1
1	12577828	12608981
2	9871975	12442063
3	15838489	12341262
4	10356967	12164392
Std. dev.	2719794	186131
COV	0.224	0.015

Table 5-3. A comparison of the generic scheduling algorithm 2 and the application-specific scheduling algorithm 2. The figures specify the amount of work (receiving points) processed by each processor

Processor Id	Generic Algorithm 2	Application-specific Algorithm 2
1	13907317	12304512
2	11857173	12206740
3	11092422	12117705
4	11788347	12016302
Std. dev.	1214175	123152
COV	0.100	0.010

The results for Algorithm 2 are shown in Table 5-3. Again, the application-specific algorithm produces a substantially better work balance among the processors. The application-specific algorithm is exactly an order of magnitude better with respect to the standard deviation and the co-efficient of variance.

Table 5-4 illustrates the results achieved with Algorithm 3 (WorkPool). Both versions of Algorithm 3 are capable of producing very good work balance results. The application-specific algorithm is slightly better in terms of the work balance. Furthermore, a comparison of the standard deviation and the co-efficient of variation indicates that Algorithm 3 produces substantially more equal work distribution than the two other algorithms.

Table 5-4. A comparison of generic scheduling algorithm 3 and the application-specific scheduling algorithm 3. The figures specify the amount of work (receiving points) processed by each processor.

Processor Id	Generic Algorithm 3	Application-specific Algorithm 3
1	12091455	12143245
2	12180042	12163003
3	12183016	12209745
4	12190746	12129266
Std. dev.	46792	35128
COV	0.0038	0.0029

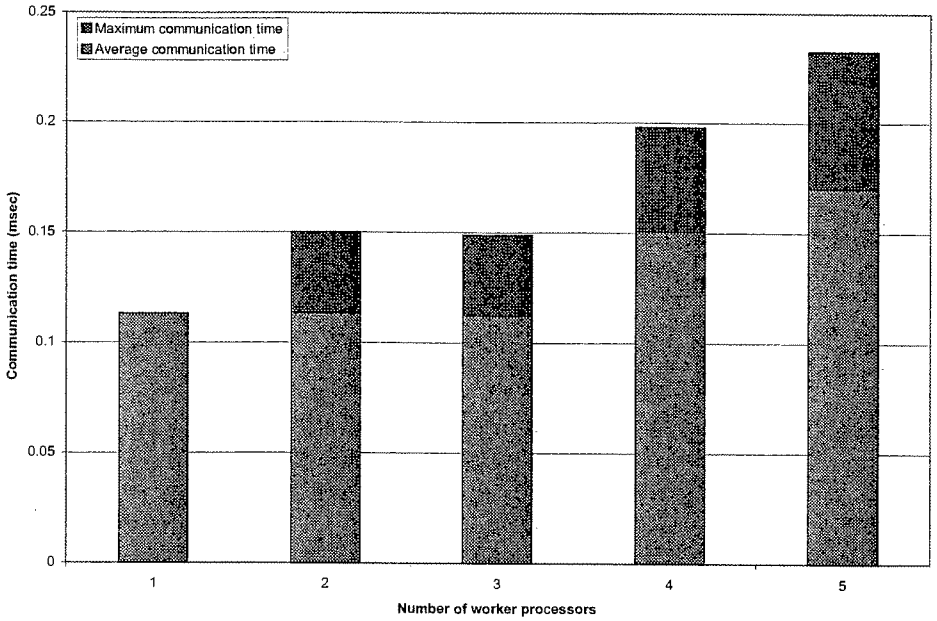


Figure 5-7. The average and maximum communication times of scheduling requests with the application-specific scheduling algorithm 3.

Very interesting results were seen with the application-specific WorkPool algorithm when the execution times of two versions of Algorithm 3 were compared. Although, the application-specific scheduling algorithm computed the most balanced distribution of workunits among the processors, the execution time of the GSM network simulator with the algorithm was longer than with the generic WorkPool algorithm. Further study showed that the application-specific WorkPool algorithm has a bottleneck on the master processor. Since the workunits are sorted and, therefore, processors have almost equal amounts of work to do, the master processor becomes congested due to a number of simultaneous requests. Figure 5-7 illustrates the average and maximum request times for the application-specific WorkPool algorithm. The average request time grows over 50% when the number of worker processors is increased from 1 to 5. The growth is even more drastic, if the maximum communication times are compared; the execution is over 105% longer with 5 worker processors than with 1 worker processor. To alleviate the problem with concurrent requests on the master processor either the order of workunits should be permuted or a pre-fetching mechanism should be implemented. However, the permutation of workunits is technically achieved by not ordering the workunits in the first place. A more sophisticated permutation method that would consider the computational requirements of the workunits was not implemented, since it would have introduced overhead. On the other hand, a pre-fetching method could have yielded improvements in the performance. However, the implementation of pre-fetching in the GSM network simulator would have been a relatively tedious task due to the structure of the simulator. The implementation would also have had to consider a random element in the timing of the pre-fetch to avoid a situation similar to the one observed in the current implementation. Since the generic version of the algorithm provided good speedups, and the main goal of the study was to research the scheduling schemes, the pre-fetch method was not

considered as an option to improve the performance (See Conclusions for the description of future work).

5.2. The WCDMA System Simulator

The WCDMA (Wide-band Code Division Multiple Access) system simulator was also developed at Nokia Research Center to study third generation (3G) mobile networks and standards, and to act as a platform for 3G algorithm studies [Oja98]. The simulator emulates a number of mobile phones in a network. The two main components of the simulator are power control operations and interference computations.

Initial tests indicated that the execution times of even relatively simple and short (in terms of simulated time) simulations were excessive, which limited the usability of the simulator. In order to obtain an adequate amount of stable results, especially for algorithm design, detailed and long simulation runs are required. Proper simulation runs led to execution times that were measured in days rather than hours. Therefore, as part of the research effort for this thesis, the execution of the sequential WCDMA system simulator was parallelized. Before the parallel implementation is discussed, the simulation model of the WCDMA simulator is explored.

5.2.1. Simulation

The simulation process of the WCDMA system is much more complex than a GSM system. For example, power control operations are performed every 0.625 milliseconds in the WCDMA, whereas the interval is 480 milliseconds in the GSM. This translates into a more time-consuming simulation process for the WCDMA system simulator. Furthermore, the WCDMA requires a number of additional operations to be performed due to interference calculation [Hut99].

The structure of the simulator is shown in Figure 5-8. The two most time-consuming parts are the terminal calculation and the interference calculation. Tests indicated that approximately 75% of the total execution time is spent in these two operations.

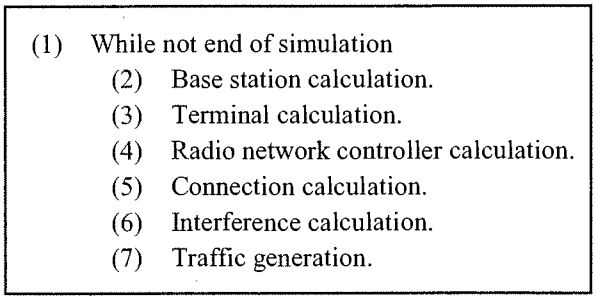


Figure 5-8. The structure of the WCDMA system simulation.

Terminal Calculation

The terminal calculation involves the processing of all active terminals (mobile phones) in the network. The pseudo-code for the terminal calculation is shown in Figure 5-9. Step 1 is required to guarantee that only active terminals are processed. In Step 2, the radio algorithms are executed for a terminal, if it has at least one non-idle connection. The terminal is moved in Step 3. Step 4 guarantees that the frame error rate (FER) calculations and power control

are performed at certain intervals for all active terminals with active connections. In Steps 5 and 6, the FERs are calculated and the power control operations are executed for the uplink direction. The power control for the downlink direction is carried out in Steps 7 and 8. The necessary measurements to determine the need for handovers are obtained in Step 9, and the handovers are performed in Step 10.

```
(1)  If Terminal Active
       (2)  If terminal's connection not silent
                  Run radio algorithms.
       (3)  Move terminal.
       (4)  If end of frame and
            connection active and
            connection not silent
              (5)  Calculate FER values.
              (6)  Do power control.
       (7)  Set power control commands.
       (8)  Do power control.
       (9)  If beginning of frame
                  Do measurements for handovers.
       (10) If handover needed
                  Do handover.
```

Figure 5-9. The structure of the terminal calculation.

Interference Calculation

The interference calculation is one of the key elements of the WCDMA system. In order to optimize the power consumption of mobile phones and to maximize the number of mobile phones in a cell, the interference has to be calculated accurately and proper actions taken based on these calculations. The interference calculation has to process all base stations and terminals in the system. Figure 5-10 shows the steps of the interference calculation. Steps 1 and 2 involve the initialization of the interference values. The loop in Step 3 goes through all base stations in the system. Step 4, in turn, goes through all the mobiles attached to a specific base station. The uplink and downlink interferences for each mobile are calculated in Steps 5 and 6. The total interferences are computed in Steps 7 and 8.

```
(1)  Set all base stations' interferences to zero.
(2)  Set all terminals' interferecences to zero.
(3)  Process each base station.
       (4)  Process each terminal .
              (5)  Calculate uplink interference.
              (6)  Calculate downlink interference.
(7)  Calculate total interference to all base stations.
(8)  Calculate total interference to all terminals.
```

Figure 5-10. The structure of the interference calculation.

5.2.2. Parallelization

A data-parallel approach was taken to run the WCDMA simulator in parallel. Due to the fact that the original implementation of the simulator was created without any consideration to parallel execution and the structure of the simulator could not be changed, the data-parallel approach proved to be the most suitable one. The initial implementation was written in C++ (deploying the object-oriented paradigm) and optimized for reuse and sequential execution. The C++ programming paradigm, with the restriction that the structure of the simulator could not be changed, made the parallelization very cumbersome. A decision was made not to transfer objects from one processor to another via a network that would require the marshaling and unmarshaling of the objects. In light of this decision the shared memory environment was the only suitable environment for parallelization.

The two previously discussed parts of the simulator were parallelized. In the terminal calculation, the workload consisted of mobile stations that were handled one at a time by a processor. There were no dependencies between mobile stations, thus no synchronization was required. In the interference calculation, a high level approach was taken; instead of distributing mobile stations among the processors, base stations were allocated to processors. A distribution based on mobile stations would have generated an excessive amount of communication and overhead.

For both parallelized parts, the WorkPool algorithm was deployed to distribute the workload. The implementations of the WorkPool algorithm had a definite advantage over the other two algorithms; the workunits (mobiles stations and base stations) did not present identical computational requirements for the processors. Thus, in order to dynamically assign workunits to processors, and maximize work balance, the WorkPool algorithm was the preferred choice.

The parallel WCDMA system simulator was implemented only for a shared memory environment due to the limited availability of other parallel environments and the timeframe in which the source code of the WCDMA simulator was available.

Shared Memory Environment

The two most time-consuming parts were parallelized with the help of threads. One set of threads was created at the beginning of the computation and they were used in both parallelized parts. For the terminal calculation, a highly application-specific scheduling algorithm was implemented; the algorithm was developed to consider the fact that data structures containing the information about the mobile stations were scattered over a large array. The algorithm was a combination of the application-specific algorithms 1 and 3. First, the algorithm distributed workunits (mobile stations) among processors based on a simple division operation. This operation considered the average number of workunits and the number of processors available. In cases where this division operation had a remainder, the remaining workunits were dynamically assigned one at a time (as in Algorithm 3) to processors that had finished their original work. A combination of two algorithms was required to achieve optimal results. The first algorithm generated less overhead but was still able to produce good work balance due to the fact that the workunits required almost an equal amount of work. Implementing the WorkPool algorithm by itself would not have been prudent. The algorithm would have introduced more overhead and, perhaps, generated a bottleneck due to a large number of simultaneous requests for work.

In the interference calculation, the workunits consisted of the base stations in the system. Each base station contained a random number of mobile stations. Therefore, the computational requirements to process workunits were not equal. This led to an implementation of the WorkPool algorithm. A synchronization method (mutual exclusion lock) was implemented to correctly serve the requests, i.e. update the index to the next unprocessed workunit and return the index. Due to the shared memory available, communication and the request-reply paradigm were handled through the memory.

5.2.3. Results

The speedups achieved are shown in Figure 5-11. Since the results are shown only for the shared memory environment, the figure also depicts the speedups achieved for the two parallelized parts rather than just the total execution time. The interference calculation, being the most computationally demanding, achieved the highest speedups; with 4 processors a speedup of 3 was obtained. The terminal calculation performed slightly worse mainly because of the small amount of work processed. The speedup of the whole simulator was substantially less than that of the two parallelized parts. The main reason for average results was the existence of a relatively large sequential part that dominated the execution times of the whole simulator.

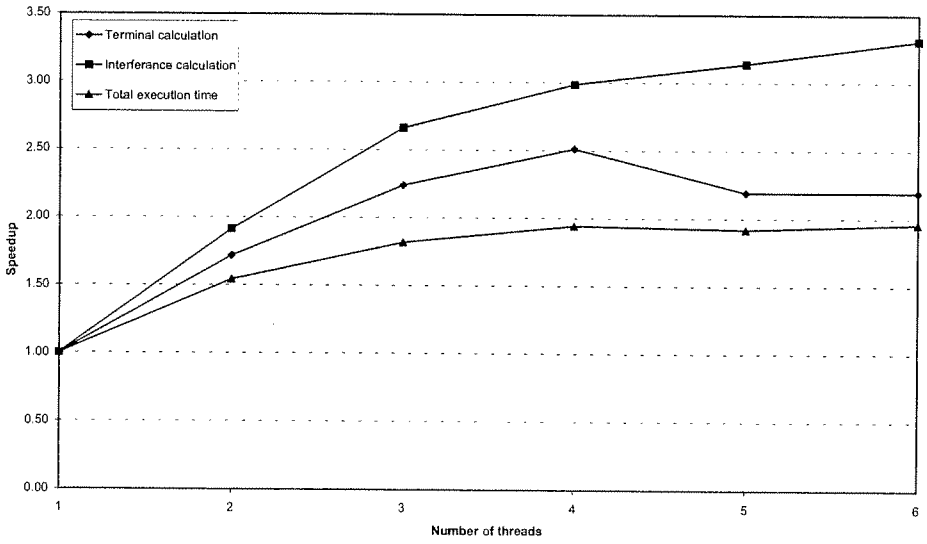


Figure 5-11. The speedups achieved in a shared memory environment with the data-parallel version of the WCDMA system simulator.

5.3. Discussion

The GSM network and WCDMA system simulators were presented in this chapter as the case studies for the scheduling algorithms developed. Both the simulators were parallelized in a data-parallel fashion. All three scheduling algorithms discussed in Chapter 3 were implemented for the GSM network simulator to study their behaviour and performance in a real life application.

The results obtained from both simulators showed that the application-specific WorkPool algorithm produced the best results. However, the application-specific WorkPool algorithm exhibited interesting results in the GSM network simulator. Although, the algorithm generated the most optimal work balance, the execution times of the GSM network simulator were not the most optimal. Further investigation showed that a bottleneck was introduced by the master processor, which caused delays on the worker processors.

Even though the WorkPool algorithm proved to be the most suitable algorithm for the two case studies it does not diminish the potentials of the other two algorithms. The superior performance of the WorkPool algorithm can be explained by looking at the characteristics of the workunits in both simulators; the workunits were vastly different in their computational requirements. The two other algorithms besides the WorkPool algorithm are more optimal in cases where the workunits are close to equal in size. This proved to be true in a case where a scheduling algorithm was designed for the terminal calculation in the WCDMA system simulator. In addition, the two scheduling algorithms do not suffer from the bottleneck caused by a master processor, since neither one has a master processor.

Chapter 6. Conclusions

Parallel computing has attracted attention during the past few decades as a vehicle for solving complex problems more efficiently. However, parallel systems introduce a number of new issues that do not exist in sequential computing, such as communication, synchronization, and scheduling. Some, if not all, of these issues have to be addressed each time a sequential application is converted to run in parallel, or a new parallel application is implemented.

The goal of this thesis was to investigate these issues and to propose solutions to obtain good performance from multiprocessor systems. Three aspects of parallel computing were studied:

- Parallel environments
- Scheduling
- Programming in SMP NOWs

Three parallel environments were identified based on memory architecture. In fact, the first two environments considered were very distinct: shared memory and distributed memory environments. The third environment was a combination of shared memory and distributed memory environments (network of workstations). First, all three environments were studied to evaluate the advantages and drawbacks of their communication networks. It was concluded that an interconnecting network plays a role in achieving good performance results in any kind of parallel environment. Furthermore, various observations were made with respect to communication libraries and their optimal working environments. The Myrinet network results were significantly below expectations due to the deployment of partially optimized message passing libraries. Unfortunately, the fully optimized libraries were not available due to technical difficulties. Overall, each of the environments exhibited characteristics that are beneficial for certain parallel applications. However, all the environments had deficiencies as well. It was concluded that no single environment was suitable for all kinds of applications merely due to hardware characteristics. Second, the environments were further examined to study their support for software. There is software, such as thread libraries, that are suitable for a shared memory environment. On the other hand, message passing libraries, like the MPI and the PVM, are available for distributed memory environments. Although the MPI supports communication through a global memory in a shared memory environment, it was observed that the performance of such an MPI library was not acceptable. A number of communication tests for different software libraries were carried out to determine the significance of the software and the interconnecting network on the performance of a parallel application.

The focus was then shifted from the environments to scheduling. Scheduling is responsible for assigning workunits to processors in a way that the work balance among processors is optimized and the amount of overhead generated is minimized. In order for a scheduling algorithm to provide the best possible work balance it required detailed information about the workunits being distributed. Therefore, a concept of application-specific information was introduced. The application-specific information contained estimates or accurate data

regarding the computational requirements of workunits that was then provided to a scheduling algorithm. With the information, a scheduling algorithm was capable of assigning workunits to processors more efficiently and effectively. Three generic scheduling algorithms were enhanced to utilize the application-specific information to investigate the benefits gained from the information. The results achieved showed significant improvements in the work balance and the performance of the applications when an application-specific scheduling algorithm was used instead of a corresponding generic algorithm. In particular, the WorkPool algorithm proved to be a very powerful mechanism in scheduling workunits among processors in any parallel environment. However, it did not come without its own deficiencies; it is possible that the master processor, that is responsible for handling workunit requests, develops a bottleneck that causes noticeable degradation in the performance.

Third, a new programming paradigm for SMP NOWs was designed and implemented. The programming paradigm called the MPIT combines the best characteristics of the MPI and threads. The MPI is used only in communication between the workstations, whereas threads are deployed to run code on processors within workstations. The MPIT programming paradigm facilitates heterogeneous SMP NOWs by allowing the number of threads created for each workstation to be configurable, as well as providing mechanisms to control the creation and termination of threads during the execution time. In addition, the MPIT has a built-in mechanism to perform scheduling for data-parallel applications according to the WorkPool algorithm developed. The scheduling occurs in the dedicated communication thread and, therefore, does not impact the performance of worker threads. The results showed that the use of the MPIT improves the performance of a conventional MPI application significantly in SMP NOWs.

In summary, the main contributions of the thesis were:

- The study of shared and distributed memory environments, networks of workstations, and the interconnecting networks used in these parallel environments.
- The design and implementation of application-specific scheduling algorithms.
- The design and implementation of the MPIT programming paradigm for networks of SMP workstations.

It can be concluded that efficient parallel computing requires that each environment is studied carefully, scheduling is handled properly with respect to the environment, and the appropriate programming paradigm is deployed. Each environment, scheduling algorithm and programming paradigm has its advantages and drawbacks. Moreover, there is no single combination of scheduling algorithm and programming paradigm that would produce optimal results in all parallel environments.

The future will show how parallel environments evolve. For now, SMP NOWs are seen as one of the most cost-efficient ways of parallel computing. Therefore, more work will be done on the MPIT to further increase its functionality and performance. In addition, scheduling in heterogeneous SMP NOWs requires more research in order to fully utilize the available resources in such environments.

Chapter 7. Summary of the Publications

This thesis includes 7 publications. The publications discuss three parallel environments, scheduling in data-parallel applications, and a programming paradigm for networks of SMP workstations. The first three publications study the environments and their suitability for parallel computation. The fourth publication introduces and discusses application-specific scheduling in data-parallel applications. The fifth publication explores the MPIT programming paradigm developed for SMP NOWs. The last two publications present the two simulators that utilize the application-specific scheduling algorithms developed for this thesis.

7.1. Publication 1

Huttunen P., Porras J., and Ikonen J.: Analysis of Parallel Environments for Mobile Network Simulation. *In Proceedings of European Simulation Symposium*, Hamburg, Germany, September 28-30, 2000, pp. 164-168.

This article evaluates the three parallel environments (shared memory, distributed memory, and network of workstations). Different message passing libraries, such as the SHMEM and the MPI, were tested in the environments. The GSM network simulator was used as the case study to compare the implementations. For a shared memory environment, the simulator was parallelized by utilizing the POSIX threads. In a distributed memory environment, the SHMEM message passing library was deployed to handle the communication among the processors. In a NOW environment, the MPI provided the communication medium for the processors to exchange messages.

7.2. Publication 2

Huttunen P., Ikonen J., and Porras J.: The Impact of Communication in Distributed Simulation. *In Proceedings of European Simulation Symposium*, Marseille, France, October 18-20, 2001, pp. 111-115.

This article focuses on interconnection networks in the distributed and NOW environments. The contribution of the paper is threefold. First, the paper discusses various reasons for communication in distributed applications (simulators). These reasons include communication, synchronization, and scheduling. Second, the most common message passing libraries are introduced and compared. The libraries discussed are the MPI, the PVM, and the TCP sockets. Third, results from the communication tests are shown and analyzed. The initial results for completion time, latency, throughput, and the impact of communication are given for the TCP and the MPI in a 100 Mbps Ethernet network. These, as well as new results, are presented in this thesis (Chapter 2).

7.3. Publication 3

Huttunen P., Porras J., and Ikonen J.: A Study of Threads and MPI libraries for Implementing Parallel Simulation. *In Proceedings of European Simulation Symposium*, Hamburg, Germany, September 28-30, 2000, pp. 96-102.

This publication studies the utilization of threads and the MPI for parallel computation. The programming paradigms are introduced by illustrating the most common communication function calls and examples. Furthermore, synchronization and scheduling in both programming paradigms are investigated. Common knowledge is that threads are useful in shared memory environments, whereas the MPI is used in distributed and NOW environments. However, some implementations of the MPI message passing library, such as *mpich*, can be optimized to communicate through the shared memory, if the two communicating processors are located in the same workstation. This article includes a study conducted to research the performance of the shared memory support for the MPI. The results shown consist of a comparison of the communication times with threads, and shared memory MPI and conventional MPI implementations.

7.4. Publication 4

Huttunen P, Ikonen J., and Porras J.: Enhancing Load Balancing in a Data-Parallel GSM Network Simulator through Application-Specific Information. *In Proceedings of Conference on Applied Parallel Computing*, Helsinki, Finland, June 15-18, 2002, pp. 542-554.

This paper introduces a new way of scheduling workunits with application-specific information. The application-specific information is additional data concerning workunits to be distributed among processors. The application-specific information is extracted by a scheduling algorithm or it is provided to the algorithm by an application. Ideally, the information provides the scheduling algorithm with adequate knowledge of the workunits (for example, computational requirements) to distribute them more effectively and efficiently. The article presents three generic scheduling algorithms that were enhanced to take advantage of the application-specific information. The results with the GSM network simulator show that the application-specific versions of all three algorithms are capable of producing better work balances and improving the performance of the application.

7.5. Publication 5

Huttunen P., Ikonen J., and Porras J.: MPIT – Communication/Computation Paradigm for Networks of SMP workstations. *In Proceedings of Conference on Applied Parallel Computing*, Helsinki, Finland, June 15-18, 2002, pp. 160-171.

This research paper presents a new programming paradigm for SMP NOWs. The paradigm, called the MPIT, combines the MPI and threads into a programming model that utilizes the best characteristics of shared and distributed environments. It creates one process with a number of threads, rather than creating numerous processes in a workstation. The threads execute the code and communicate with each other through the memory allocated for the process. The communication among the workstations is handled by a separate communication thread with the help of the MPI. This frees the worker threads to process their data without having to stop to perform communication, which, in turn, decreases the communication-to-computation ratio. Furthermore, the MPIT programming paradigm includes a mechanism for

performing scheduling. The initial results indicate that combining the two programming paradigms and utilizing a dedicated communication thread can improve the performance of a conventional MPI application substantially in an SMP NOW.

7.6. Publication 6

Porras J., Huttunen P., Ikonen J.: Accelerating Ray Tracing Based Cellular Radio Coverage Calculation by Parallel Computing Techniques. *Annual Review of Communications*, Vol. 53, 2000.

This paper introduces the GSM network simulator. The internal implementation of the simulator is explored to the extent that the workload generated by the simulator can be understood. The parallelization process is described for shared and distributed memory environments. Furthermore, detailed results from the parallel environments are given with performance analysis.

7.7. Publication 7

Huttunen P., Ikonen J., and Porras J.: Parallelization of a WCDMA System Simulator for a Shared Memory Multiprocessor Machine. *In Proceedings of European Simulation Symposium*, Erlangen-Nuremberg, Germany, October 26-28, 1999, pp. 556-560.

This publication discusses the WCDMA system simulator. A detail description of the simulator is given to show the need for parallelization and how the parallelization can be done. The parallel implementation of the system simulator with the WorkPool scheduling algorithm in a shared memory environment is presented. The results achieved show that performance improves over the sequential WCDMA system simulator. However, the study also points out deficiencies in the sequential implementation that prevents more optimal parallelization. The WCDMA system simulator is a good example of an application that was not originally designed with parallel execution in mind. Therefore, the structure of the simulator and the requirement that the structure not change did not allow for the best possible parallel implementation.

7.8. Errata

In Publication 4, the source of the reference 7 should say "In Proceedings of Computer Conference" rather than "In Proceedings of XXX".

Bibliography

[Ach97] Acharya A., Edjlali G., and Saltz J.: The Utility of Exploiting Idle Workstations of Parallel Computation. *In Proceedings of SIGMETRICS*, 1997, pp. 225-234.

[Adl95] Adler M., Chakrabarti S., Mitzenmacher M., and Rasmussen L.: Parallel Randomized Load Balancing. *In Proceedings of Symposium on Theory of Computing*, 1995, pp. 238-247.

[And97] Anderson E.C., Brooks J.P., Gassi C.M., and Scott S.L.: Performance Analysis of the T3E Multiprocessor. *In Proceedings of Conference on Supercomputing*, 1997.

[Ayg99] Ayguadé E., Martorell X., Labarta J., Gonzàles M., and Navarro N.: Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. *In Proceedings of Conference on Parallel Processing*, 1999, pp. 172-180.

[Bad97] Bader D.A., and Jájá J.: SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, Vol. 58, No. 1, 1997, pp. 92-108.

[Bar99] Barbosa D.M., Kitajima J.P., and Weira W.: Parallelizing MPEG Video Encoding Using Multiprocessors. *In Proceedings of Computer Graphics and Image Processing*, 1999, pp. 215-222.

[Ben00a] Bennett B.H., Davis E., and Kunau T.: Beowulf Parallel Processing for Dynamic Load-balancing. *In Proceedings of Aerospace Conference*, 2000, pp. 389-395.

[Ben00b] Bender M.A., Rabin M.O.: Scheduling Cilk Multithreaded Parallel Programs on Processors of Different Speeds. *In Proceedings of Symposium on Parallel algorithms and architectures*, 2000, pp. 13-21.

[Ber96] Berman F., Wolski R., Figueira S., Schpf J., and Shao G.: Application-Level Scheduling on Distributed Heterogeneous Networks. *In Proceedings of Supercomputing Conference*, 1996, pp. 1-28.

[Ber99] Berenbrink P., Friedetzky T., and Steger A.: Randomized and Adversarial Load Balancing. *In Proceedings of Symposium on Parallel Algorithms and Architectures*, 1999, pp. 175-184.

[Boy02] Boyd T., and Dasgupta P.: Process migration: A Generalized Approach Using a Virtualizing Operating System. *In Proceedings of Conference on Distributed Computing Systems*, 2002, pp. 348-355.

[Bre99] Brest J., Zumer V., and Ojstersek M.: Dynamic Scheduling on a PC Cluster. *In Proceedings of Symposium on Applied Computing*, 1999, pp. 496-500.

[Bru97a] Bruck J., Dolev D., Ho C-T., Rosu M-C., and Strong R.: Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. *Journal of Parallel and Distributed Computing*, Vol. 40, 1997, pp. 19-34.

[Bru97b] Brunstrom A., and Simha R.: Dynamic versus Static Load Balancing in a Pipeline Computation. *Journal of Modelling and Simulation*, Vol. 17, No. 4, 1997, pp. 317-327.

[Bur94] Burns G.D., Daoud R.B., and Vaigl J.R.: LAM: An Open Cluster Environment for MPI. *In Proceedings of Supercomputing Symposium*, 1994, pp. 379-386.

[But97] Butenhof D.R.: Programming with POSIX Threads. Addison-Wesley, 1997.

[Büc02] Bücker H.M., Lang B., Rasch A., Bischof C.H., and an Mey D.: Explicit Loop Scheduling in OpenMP for Parallel Automatic Differentiation. *In Proceedings of Symposium on High Performance Computing Systems and Applications*, 2002, pp. 112-117.

[Cap00] Cappelo F., and Etiemble D.: MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmark. *In Proceedings of Conference on Supercomputing*, 2000, Article No. 12.

[Cat01] Catarinucci L., Palazzari P., and Tarricone L.: Parallel Simulation of Radio-Base Antennas on Massively Parallel Systems. *In Proceedings of Symposium on Parallel and Distributed Processing*, 2001.

[Cav01] Cavalheiro G.G.H.: A General Scheduling Framework for Parallel Execution Environments. *In Proceedings of Symposium on Cluster Computing and Grid*, 2001, pp. 680-687.

[Cha02] Chaver D., Prieto M., Pinuel L., and Tirado F.: Parallel Wavelet Transform for Large Scale Image Processing. *In Proceeding of Symposium on Parallel and Distributed Processing*, 2002, pp. 29-34.

[Chi98] Chiang C-L., Wu J-J., Lin N-W.: Toward Supporting Data Parallel Programming on Clusters of Symmetric Multiprocessors. *In Proceedings of Conference on Parallel Distributed Systems*, 1998, pp. 607-614.

[Chi99] Chiola G., and Ciaccio G.: Lightweight Messaging Systems. *High Performance Cluster Computing*, Vol. 1, Prentice Hall, 1999, pp. 246-269.

[Cor99] Corradi A., Leonardi L., and Zambonelli F.: Diffusive Load-Balancing Policies for Dynamic Applications. *Concurrency*, Vol. 7, No. 1, 1997, pp. 22-31.

[Cre02] Cremonesi P.C., and Gennaro C.: Integrated Performance Models for SPMD Applications and MIME Architectures. *Transactions on Parallel and Distributed Sytems*, Vol. 13, No. 7, 2002, pp. 745-757.

[Cru01] Cruz J., and Kihong P.: Towards Communication-Sensitive Load Balancing. *In Proceedings of Conference on Distributed Computing Systems*, 2001, pp. 731-734.

[Dai00] Dail H., Obertelli G., Berman F., Wolski R., and Grimshaw A.: Application-Aware Scheduling of a Magnetohydrodynamics Application in the Legion Metasystem. *In Proceedings of Heterogeneous Computing Workshop*, 2000, pp. 216-228.

[Das97] Dasgupta P., Majumder A.K., and Bhattacharya P.: V_THR: An Adaptive Load Balancing Algorithm. *Journal of Parallel and Distributed Computing*, Vol. 42, 1997, pp. 101-108.

[DaS01] Da Silva F.A.B., and Scherson I.D.: Simulation-based Average Case Analysis for Parallel Job Scheduling. *In Proceedings of Simulation Symposium*, 2001, pp. 15-24.

[Den01] Deng Y., and Korobka A.: The Performance of a Supercomputer Built with Commodity Components. *Parallel Computing*, Vol. 27, 2001, pp. 91-108.

[Dev00] Devine K., Hendrickson B., Boman E., St.John M., and Vaughan C.: Design of Dynamic Load-Balancing Tools for Parallel Applications. *In Proceedings of the 2000 international conference on Supercomputing*, 2000, pp. 110-118.

[Dre98a] Dreier B., Zahn M., and Ungerer T.: Parallel and Distributed Programming with Pthreads and Rthreads. *In Proceedings of Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998, pp. 34-41.

[Dre98b] Dreier B., Zahn M., and Ungerer T.: The Rthreads Distributed Shared Memory System. *In Proceedings of Conference on Massively Parallel Computer Systems*, 1998.

[Els00] Elsasser R., Monien B., and Preis R.: Diffusive Load Balancing Schemes on Heterogeneous Networks. *In Proceedings of Parallel Algorithms and Architectures*, 2000, pp. 30-38.

[Fah95] Fahringer T., Haines M., and Mehrotta P.: On the Utility of Threads for Data Parallel Programming. *In Proceedings of Conference on Supercomputing*, 1995, pp. 51-59.

[Fer01] Ferreira R., Agrawal G., and Saltz J.: Compiler and Runtime Analysis for Efficient Communication in Data Intensive Applications. *In Proceedings of Parallel Architectures and Compilation Techniques*, 2001, pp. 231-242.

[Feu94] Feuerstein M.J., Blackard K.L., Rappaport T.S., Seidel S.Y., and Xia H.H.: Path Loss, Delay Spread, and Outage Models as Functions of Antenna Height for

Microcellular System Design. *Transactions on Vehicular Technology*, Vol. 43, No. 3, 1994, pp. 487-498.

[Fle99] Fleury M, Downton A.C., and Clark A.F.: Scheduling Schemes for Data Farming. *In Proceedings of Computers and Digital Techniques*, Vol. 146, No. 5, 1999, pp. 227-234.

[Fos96] Foster I., Kesselman C., and Tuecke S.: The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, Vol. 37, 1996, pp. 70-82.

[Gan96] Ganesan R., Govindarajan K., and Wu M-Y.: Comparing SIMD and MIMD Programming Models. *Journal of Parallel and Distributed Computing*, Vol. 35, 1996, pp. 91-96.

[Gan00] Gan B.P., Low Y.H., Jain S., Turner S.J., Cai W., Hsu W.J., and Huang S.Y.: Load Balancing for Conservative Simulation on Shared Memory Multiprocessor Systems. *In Proceedings of Workshop on Parallel and Distributed Simulation*, 2000, pp. 139-146.

[Gei96] Geist A., Beguelin A., and Dongarra J.: PVM: Parallel Virtual Machine. MIT Press, 1996.

[Ger91] Gerndt M.: Work Distribution in Parallel Programs for Distributed Memory Multiprocessors. *In Proceedings of Conference on Supercomputing*, 1991, 96-104.

[Gho94] Ghosh B., and Muthukrishnan S.: Dynamic Load Balancing in Parallel and Distributed Networks by Random Matchings. *In Proceedings of Symposium on Parallel Algorithms and Architectures*, 1994, 226-235.

[Gho99] Ghosh B., Leighton F.T., Maggs B.M, Muthukrishnan S., Plaxton C.G., Rajaraman R., Richa A.W., Tarjan R.E., and Zuckerman D.: Tight Analyses of Two Local Load Balancing Algorithms. *Journal on Computing*, Vol. 29, No. 1, 1999, pp. 29-64.

[Gri94] Grimshaw A.S., Weissman J.B., West E.A., and Loyot E.C.: Metasystems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, Vol. 21, 1994, pp. 257-270.

[Gro96] Gropp W., Lusk E., and Skjellum A.: High-performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, Vol. 22, No. 6, 1996, pp. 789-828.

[Gro99] Gropp W., Lusk E., and Skjellum A.: Using MPI – Portable Parallel Programming with Message Passing Interface. MIT Press, 1999.

[Had02] Hadjidoukas P.E., Polychronopoulos E.D., and Papatheodorou T.S.: Integrating MPI and Nanothreads Programming Model. *In Proceedings of Euromicro*

Workshop on Parallel, Distributed and Network-based Processing, 2002, pp. 309-316.

[Hai94] Haines M., Cronk D., and Mehrotra P.: On the Design of Chant: A Talking Threads Package. *In Proceedings of Conference on Supercomputing*, 1994, pp. 350-359.

[Hei96] Heiska K., and Kangas A.: Microcell Propagation Model for Network Planning. *In Proceedings of Symposium on Personal, Indoor, and Mobile Radio Communications*, 1996, pp. 148-152.

[Hen98] Henrichs J.: Optimizing and Load Balancing Metacomputing Applications. *In Proceedings of Conference on Supercomputing*, 1998, pp. 165-171.

[Hen00] Henty D.S.: Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. *In Proceedings of Conference on Supercomputing*, 2000, Article No. 4.

[Hey98] Heymann E., Tinetti F., Luque E.: Preserving Message Integrity in Dynamic Process Migration. *In Proceedings of Workshop on Parallel and Distributed Processing*, 1998, pp. 373 –381.

[Hog99] Hoganson K.E.: Workload Execution Strategies and Parallel Speedup on Clustered Computers. *Transactions on Computers*, Vol. 48, No. 11, 1999, pp. 1173-1182.

[Hsi00] Hsieh J., Leng T., Mashayekhi V., and Rooholamini R.: Architectural and Performance Evaluation of GigaNet and Myrinet Interconnects on Clusters of Small-Scale SMP Servers. *In Proceedings of Conference on Supercomputing*, 2000, Article No. 18.

[Hu99] Hu Y.C., Lu H., Cox A.L., Zwaenepoel W.: OpenMP for Networks of SMPs. *In Proceedings of Symposium on Parallel and Distributed Computing*, 1999, pp. 302-310.

[Hui99] Hui C-C., and Chanson S.T.: Hydrodynamic Load Balancing. *Transactions on Parallel and Distributed Systems*, Vol. 10, No. 11, 1999, pp. 1118-1136.

[Hut98] Huttunen P., Porras J., Ikonen J., and Sipilä K.: Parallelization of Propagation Model Simulation. *In Proceedings of European Simulation Symposium*, 1998, pp. 321-325.

[Hut99] Huttunen P.: Improving the Performance of a WCDMA System Simulator through Parallel Computing Techniques. *Master's Thesis*, Lappeenranta University of Technology, Finland, 1999.

[Hut00] Huttunen P., Ikonen J., and Porras J.: Parallel Simulation of a GSM Network on a Cluster of Workstations. *In Proceedings of European Simulation Multiconference*, 2000, pp. 563-567.

[Hwa98] Hwang K., and Xu X.: Scalable Parallel Computing: Technology, Architecture, Programming. WCB/McGraw-Hill, 1998.

[Iko01] Ikonen J.: Improving Distributed Simulation in a Workstation Environment. *Doctorate Thesis*, Lappeenranta University of Technology, Finland, 2001.

[Isl97] Islam N.: Customized Message Passing. *Journal of Parallel and Distributed Computing*, Vol. 41, 1997, pp. 205-224.

[Jon00] Jones K.G., and Das S.R.: Parallel Execution of a Sequential Network Simulator. *In Proceedings of Simulation Conference*, 2000, pp. 418-424.

[Kaf95] Kafura D., and Huang L.: MPI++: A C++ Language Binding for MPI. *In Proceedings of MPI Developers Conference*, 1995.

[Kai99] Kai M., and Shimada M.: Task Scheduling Algorithms Based on Heuristic Search Taking Account of Communication Overhead. *In Proceedings of Conference on Communications, Computers and Signal Processing*, 1999, pp. 145-150.

[Kat97] Katevenis M.G.H., Markatos E.P., Kalokerinos G., and Dollas A.: Telegraphos: A Substrate for High-Performance Computing on Workstation Clusters. *Journal of Parallel Computing*, Vol. 43, 1997, pp. 94-108.

[Kim01] Kim S.C., and Lee S.: Measurement and Prediction of Communication Delay in Myrinet Networks. *Journal of Parallel and Distributed Computing*, Vol. 61, 2001, pp. 1692-1704.

[Kin88] King C-T., Chou W-H., and Ni L.M.: Pipelined Data Parallel Algorithms – Concept and Modeling. *In Proceedings of Conference on Supercomputing*, 1988, pp. 385-395.

[Kle96] Kleiman S., Shah D., and Smaalders B. Programming with Threads. SunPress, 1996.

[Kos00] Koski K., and Fagerholm J.: Report of the Pilot Project for PC Clusters. Center for Scientific Computation, Finland, 2000, http://www.csc.fi/reports/pc-cluster/.

[Kul00] Kulkarni P., and Sengupta I.: A New Approach for Load Balancing Using Differential Load Measurement. *In Proceedings of Conference on Information Technology: Coding and Computing*, 2000, pp. 355-359.

[Kum01] Kumar V.: Introduction to Parallel Computing. Addison-Wesley, 2001.

[Kur00] Kurjenniemi J., Hämäläinen S., and Ristaniemi T.: System Simulator for UTRA TDD. *In Proceedings of CDMA International Conference & Exhibition*, 2000, pp. 370-374.

[Lau97] Lauria M., and Chien A.: MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, Vol. 40, 1997, pp. 4-18.

[Lei99] Leinberger W., Karypis G., Kumar V.: Job Scheduling in the Presence of Multiple Resource Requirements. *In Proceedings of Conference on Supercomputing*, 1999, Article No. 47.

[Lew96] Lewis B., and Berg D.J. Threads Primer: A Guide to Multithreaded Programming. SunPress, 1996.

[Li98] Li K.: Deterministic and Randomized Algorithms for Distributed On-line Task Assignment and Load Balancing without Load Status Information. *In Proceedings of Symposium on Applied Computing*, 1998, pp. 613-622.

[Lim99] Lim A.W., Cheong G.I., and Lam M.S.: An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication. *In Proceedings of Conference on Supercomputing*, 1999, pp. 228-237.

[Liu99] Liu Y.-L., Cheng H.-Y., and King C.-T.: High Performance Computing on Networks of Workstations through the Exploitation of Functional Parallelism. *Journal of System Architectures: The EUROMICRO Journal*, 1999, pp. 1307-1321.

[Lue99] Luecke G.R., Raffin B., and Coyle J.J.: Comparing the Communication Performance and Scalability of a SGI Origin 2000, a Cluster of Origin 2000's and a Cray T3E-1200 Using SHMEM and MPI Routines. *Journal of Performance Evaluation and Modeling for Computer Systems*, 1999.

[Mac00] Mache J., Lo V., and Garg S.: Job Scheduling that Minimizes Network Contention Due to Both Communication and I/O. *In Proceedings of Symposium on Parallel and Distributed Processing*, 2000, pp. 457-463.

[McC94] McColl W.F.: BSP Programming. *In Proceedings of DIMACS Workshop on Specification of Parallel Algorithms*, 1994, pp. 25-35.

[Mey97] Meyer T.E., Davis J.A., and Davidson J.L.: Analysis of Load Average and Its Relationship to Program Run Time on Networks of Workstations. *Journal of Parallel and Distributed Computing*, Vol. 44, 1997, pp. 141-146.

[Mic01] Michailidis P.D., and Margaritis K.G.: Parallel Text Searching Application on a Heterogeneous Cluster of Workstations. *In Proceedings of Workshop on Parallel Processing*, 2001, pp. 169 -175

[Mit97] Mitzenmacher M.: On the Analysis of Randomized Load Balancing Scheme. *In Proceedings of Symposium on Parallel Algorithms and Architectures*, 1997, pp. 292-301.

[MPI95] MPI Forum: MPI-1.1 Standard. 1995, http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[MPI97] MPI Forum: MPI-2 Standard. 1997, http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[Muk99] Mukherjee N., and Gurd, J.R.: A Comparative Analysis of Four Parallelisation Schemes. *In Proceedings of Conference on Supercomputing*, 1999, pp. 278-285.

[Myr00] Myricom Inc.: The GM Message Passing Library. 2000, http://www.myri.com/scs/GM/doc/gm.pdf.

[Nag99] Nagendra B., and Rzymianowicz L.: High Speed Networks. *High Performance Cluster Computing*, Vol. 1, Prentice Hall, 1999, pp. 201-245.

[Nie01] Nieplocha J., Ju J., and Apra E.: One-Sided Communication on the Myrinet-Based SMP Clusters using the GM Message-Passing Library. *In Proceedings of Parallel and Distributed Processing Symposium*, 2001, pp. 1707-1716.

[Nik00] Nikolopoulos D.S., Papatheodorou T.S., Polychronopoulos C.D., Labarta J., and Ayguade E.: Is Data Distribution Necessary in OpenMP? *In Proceedings of Conference on Supercomputing*, 2000, Article No. 47.

[Nor93] Norman M.G., and Thanisch P.: Models of Machines and Computation for mapping in Multicomputers. *Computing Surveys*, Vol. 25, No. 3, 1993, pp. 263-302.

[Nor96] Norton S., and Dipasquale M.D. Thread Time: The Multithreaded Programming Guide. Prentice Hall, 1996.

[Oja98] Ojanperä T., and Prasad R.: Wideband CDMA for Third Generation Mobile Communications. Artech House, 1998.

[Par99] Parab N., and Raghvedran M.: Active Messages. *High Performance Cluster Computing*, Vol. 1, Prentice Hall, 1999, pp. 270-300.

[Pla94] Plata O., and Rivera F.F.: Combining Static and Dynamic Scheduling on Distributed-Memory Multiprocessors. *In Proceedings of Conference on Supercomputing*, 1994, pp 186-195.

[Por95] Porras J., Harju J., and Ikonen J.: Parallel Simulation of Mobile Communication Networks Using a Distributed Workstation Environment. *In Proceedings of Eurosim Conference*, 1995, pp. 571-576.

[Por98a] Porras J., Ikonen J., and Harju J.: Applying a Modified Chandy-Misra Algorithm to the Distributed Simulation of a Cellular Network. *In Proceedings of Conference on Parallel and Distributed Simulation*, 1998, pp. 188-195.

[Por98b] Porras J., Ikonen J., and Harju J.: Computing the Critical Time for a Cellular Network Simulation on a Cluster of Workstations. *In Proceedings of European Simulation Multiconference*, 1998, pp. 16-19.

[Por98c] Porras J.: Developing a Distributed Simulation Environment on a Cluster of Workstations. *Doctorate Thesis*, Lappeenranta University of Technology, Finland, 1998.

[Pra97] Prakash S.R., and Srikant Y.N.: Communication Cost Estimation and Global Data Partitioning for Distributed Memory Machines. *In Proceedings of High Performance Computing*, 1997.

[Pro01] Protopopov B.V., and Skjellum A.: A Multi-Threaded Message Passing Interface (MPI) Architecture: Performance and Program Issues. *Journal of Parallel and Distributed Computing*, Vol. 61, 2001, pp. 449-466.

[Qin97] Qin X., and Baer J-L.: A Performance Evaluation of Cluster Architectures. *In Proceedings of SIGMETRICS*, 1997, pp. 237-247.

[Rap96] Rappaport T.S., Muhamed R., and Kapoor V.: Propagation Models. *The Mobile Communication Handbook*, CRC Press, 1996, pp. 355-369.

[Ren01] Rencuzogullari U., and Dwarkadas S.: Dynamic Adaptation to Available Resources for Parallel Computing in an Autonomous Network of Workstations. *In Proceedings of Symposium on Principles and Practice of Parallel Programming*, 2001, pp. 72-81.

[Ris02] Rischmuller V., Kurz S., and Rucker W.M.: Parallelization of Coupled Differential and Integral Methods Using Domain Decomposition. *Transactions on Magnetics*, Vol. 38, No. 2, 2002, pp. 981-984.

[Roh96] Roh L., Najjar W.A., Shankar B., and Böhm A.P.: Generation, Optimization, and Evaluation of Multithreaded Code. *Journal of Parallel and Distributed Computing*, Vol. 32, 1996, pp. 188-204.

[Ros91] Ross K.W., and Yao D.D.: Optimal Load Balancing and Scheduling in a Distributed Computer System. *Journal of ACM*, Vol. 38, No. 3, 1991, pp. 676-690.

[Răd01] Rădulescu A., and van Gemund A.J.C.: A Low-Cost Approach Towards Mixed Task and Data Parallel Scheduling. *In Proceedings of Conference on Parallel Processing*, 2001, pp. 69-76.

[Sav99] Savarese D.F., and Sterling T.: Beowulf. *High Performance Cluster Computing*, Vol. 1, Prentice Hall, 1999, pp. 625-645.

[Sch95] Schlagenhaft R., Ruhwandl M., Sporrer C., and Bauer H.: Dynamic Load Balancing of a Multi-Cluster Simulator on a Network of Workstations. *In Proceedings of Workshop on Parallel and Distributed Simulation*, 1995, pp. 175-180.

[Sco96] Scott S.L.: Synchronization and Communication in the T3E Multiprocessor. *In Proceedings of Conference on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 26-36.

[Sil99] Silicon Graphics Inc.: Message Passing Toolkit: PVM Programmer's Manual. 1999, http://www.fz-juelich.de/zam/docs/CrayDoc/manuals/007-3686-002/007-3686-002-manual.pdf.

[Sip96] Sipilä K., and Heiska K.: Can Ray Tracing Be Used as a Fading Generator in Simulating Micro Cellular Mobile Radio Systems? *In Proceedings of Conference on Wireless Communication*, 1996.

[Sis99] Sistare S., vande Vaart R., Loh E.: Optimization of MPI Collectives on Clusters of Large-Scale SMP's. *In Proceedings of Conference on Supercomputing*, 1999, Article No. 23.

[Ski98] Skillicorn D.B., and Talia D.: Models and Languages for Parallel Computation. *Computing Surveys*, Vol. 30, No. 2, 1998, pp.123-169.

[Skj94] Skjellum A., Doss N.E., Viswanathan K., Chowdappa A., Bangalore P.V.: Extending the Message Passing Interface. *In Proceedings of Scalable Parallel Libraries Conference II*, 1994.

[Soh96] Sohn A., Biswas R., and Simon H.D.: A Dynamic Load Balancing Framework for Unstructured Adaptive Computations on Distributed-Memory Multiprocessors. *In Proceedings of Symposium on Parallel Algorithms and Architectures*, 1996, pp. 189-192.

[Soh97] Sohn A., Sato M., Yoo N., and Gaudiot J-L.: Data and Workload Distribution in a Multithreaded Architecture. *Journal of Parallel and Distributed Computing*, Vol. 40, 1997, pp. 256-264.

[Spr01] Springer P.L.: PVM Support for Clusters. *In Proceedings of Conference on Cluster Computing*, 2001, pp. 183-186.

[Squ96] Squyres J.M., McCandless B.C., and Lumsdaine A.: Object Oriented MPI. *CSE Technical Report TR96-10*, University of Notre Dame, 1996.

[Ste97] Steenkiste P.: A High-Speed Network Interface for Distributed-Memory Systems: Architecture and Applications. *Transactions on Computer Systems*, Vol. 15, No. 1, 1997, pp. 75-109.

[Sto01] Stone J., and Ercal F.: Workstation Clusters for Parallel Computing. *Potentials*, Vol. 20, No. 2, 2001, pp. 31-33.

[Sub02] Subramani V., Kettimuthu R., Srinivasan S., and Sadayappan P.: Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests. *In Proceedings of Symposium on High Performance Distributed Computing*, 2002, pp. 359-366.

[Sun90] Sunderam V.S.: PVM: A Framework for Parallel Distributed Computing. *Concurrency: Theory and Practice*, Vol. 2, No. 4, 1990, pp. 315-339.

[Tan99] Tanaka Y., Matsuda M., Kubota K., Sato M.: COMPaS: A Pentium Pro PC-Based SMP Cluster. *High Performance Cluster Computing*, Vol 1, Prentice Hall, 1999, pp. 661-681.

[Tan00] Tang X., and Chanson S.T.: Optimizing Static Job Scheduling in a Network of Heterogeneous Computers. *In Proceedings of Conference on Parallel Processing*, 2000, 373-382.

[Tan01] Tang H., and Yang T.: Optimizing Threaded MPI Execution on SMP Clusters. *In Proceedings of Conference on Supercomputing*, 2001, pp. 381-392.

[Tha01] Thanalapati T., Dandamudi S.: An Efficient Adaptive Scheduling Scheme for Distributed Memory Multiprocessors. *Transactions on Parallel and Distributed Systems*, Vol. 12, No. 7, 2001, pp. 758-768.

[Top02] University of Mannheim, and University of Tennessee: TOP500 List, 2002 (June), http://www.top500.org/lists/2002/06/.

[Uth02] Uthayopas P., Angskun T., Maneesilp J.: On the Building of the Next Generation Integrated Environment for Beowulf Clusters. *In Proceedings of Symposium on Parallel Architectures, Algorithms, and Networks*, 2002, pp. 159-164.

[Wag97] Wagner A.S., Sreekantaswamy H.V., and Chanson S.T: Performance Models for the Processor Farm Paradigm. *Transactions on Parallel and Distributed Systems*, Vol. 8, No. 5, 1997, pp. 475-489.

[Wal98] Wallcraft A.J.: A Comparison of Several Scalable Programming Models. *In Proceedings of Workshop on Software Engineering and Code Design in Parallel Meteorological and Oceanographic Applicaations*, 1998, pp. 183-198.

[Wea99] Weaver A.C.: Xpress Transport Protocol. *High Performance Cluster Computing*, Vol. 1, Prentice Hall, 1999, pp. 301-316.

[Wil96] Wilson L.F., and Nicol D.M.: Experiments in Automated Load Balancing. *In Proceedings of Workshop on Parallel and Distributed Simulation*, 1996, pp. 4-11.

[Wil98] Wilson L.F., and Shen W.: Experiments in Load Migration and Dynamic Load Balancing in SPEEDES. *In Proceedings of Winter Simulation Conference*, 1998, pp. 483-490.

[Wu98] Wu J-J., and Liu P.: Distributed Data Structure Design for Scientific Computations. *In Proceedings of Conference on Supercomputing*, 1998, pp. 227-234.

[Zak97] Zaki M.J., Li W., and Parthasarathy S.: Customized Dynamic Load Balancing for a Network of Workstations. *Journal of Parallel and Distributed Computing*, Vol. 43, 1997, pp. 156-162.

[Zha00] Zhang Y., Sivasubramaniam A., Moreira J., and Franke H.: A Simulation-based Study of Scheduling Mechanisms for a Dynamic Cluster Environment. *In Proceedings of Conference on Supercomputing*, 2000, pp. 100-109.

Publications

Publication 1

Huttunen P., Porras J., and Ikonen J.: Analysis of Parallel Environments for Mobile Network Simulation. In Proceedings of European Simulation Symposium, Hamburg, Germany, September 28-30, 2000, pp. 164-168.

Analysis of Parallel Environments for Mobile Network Simulation

Pentti Huttunen, Jari Porras, and Jouni Ikonen
Lappeenranta University of Technology
P.O. Box 20, FIN-53851 Lappeenranta, Finland
{ Pentti.Huttunen, Jari.Porras, Jouni.Ikonen }@lut.fi

KEYWORDS
Parallel simulation, shared memory environment, distributed memory environment, cluster of workstations.

ABSTRACT

This paper compares three parallel environments: shared memory, distributed memory, and cluster of workstations. A GSM network simulator is implemented with the most optimal work balancing algorithm into each of the three environments. Benefits and drawbacks of each environment are explained regarding the network simulation. The achieved speedups from the simulation are presented. Finally, the conclusions are drawn with respect to scalability and cost efficiency of the environments.

INTRODUCTION

During the last few decades the need for increased computing power has risen drastically. A large number of computer vendors have introduced new systems based on a variety of different architectural models. Each architectural platform offers an optimal environment for specific types of computing tasks. At the present time parallel environments can be divided into three distinctive systems:

- Shared memory machines
- Distributed memory machines
- Cluster of workstations

Due to the divergence of the systems, it is essential to select the correct environment in which to run the parallel program. In theory, a program can be run in any of the three environments. However, usually there is an optimal environment for each program, where overhead is minimized, and performance is maximized. The shared memory environment enables the use of the memory as a communication medium for processors to transfer data. Thus, the communication overhead is minimal. In the distributed and cluster environments, communication has to be implemented explicitly with message-passing interface. However, the distributed nature of the environments enables the use of a considerably larger number of processors than in the shared memory environment. Since all communication in the shared memory environment is handled through the memory via the memory bus, if the number of processors rises over 10, congestion in the memory bus decreases performance considerably (Wilkinson and Allen 1999). Therefore, when the parallel application does not require constant communication between processors, distributed and cluster environments provide suitable platforms for large parallel applications. The selection of the environment depends on several factors. In general, the structure of the program dictates the optimal environment. The structure of the program reveals the need for communication, synchronization, and requirements for a work balancing algorithm.

The paper is structured as follows: Firstly, a brief introduction of the GSM network simulator is given. Secondly, all environments are described with their special features, advantages and drawbacks. In addition, the parallelization operations done for each environment are described. Thirdly, test environments, parameters, and the achieved results are presented. Finally, the conclusions are drawn, and the implementation issues specific for each environment are discussed and compared.

THE GSM NETWORK SIMULATOR

The simulator was originally designed and implemented at Nokia Research Center in Finland. The first implementation did not include any parallel code. During the test runs, it became apparent that the interactive use of the simulator required a substantial increase in computing power. Fortunately, the structure of the simulators was suitable for parallel execution.

The GSM network simulator is used to calculate a coverage generated by a base station. The coverage denotes an area in which a mobile phone can be used, i.e. the mobile station receives signals from the base station, which are strong enough for transmission of speech or data. The main purpose of the simulator is to offer a low-cost design tool for network providers. Coverage calculation is based on a ray tracing of signals from the base station. The method of ray tracing is similar to ray tracing used in 3D graphics. However, the implementation of the ray tracing differs significantly.

The area of the calculated coverage is called a map. A grid is created over the map dividing it into receiving points that are squares. The size of the map is usually 1-3 km^2, and the size of each receiving point square is from 1 m^2 to 16 m^2. The size of the receiving point has a major effect on the complexity of calculation. The basic implementation of ray tracing requires tracing of all signals from the base station, until the powers of the signals go below a set threshold. The powers of signals are reduced based on the distance and the obstacles on the route of the signals (Hata 1980; Sipilä and Heiska 1996). The field strength of each receiving point is updated when a signal arrived to the receiving point. From the laws of physics the signal can (Rappaport et al. 1996):

- propagate in free space
- reflect from an obstacle
- diffract from an edge of an obstacle

The signal suffers propagation loss due to the distance it traverses. The loss is proportional to the square of the propagated distance. Reflection takes place when the ray hits an obstacle, i.e. a building. The ray reflects from the surface, while parts of it goes through the obstacle. The simulator only recognizes the reflected signal, which suffers a loss of power due to the reflection. Diffraction happens when a ray arrives to a corner of an obstacle. Multiple new rays are created, since the ray disperses to all directions. Thus, the complexity of the calculation is increased whenever diffraction occurs. The signal propagation can be a problem, since numerous rays are launched from the base station requiring a lot of work. Nevertheless, to achieve accurate results, the number of launched rays has to be high. The complexity of the simulation increased exponentially when more signals are traced.

Since the number of rays to be traced during simulation is very high, the basic ray tracing method was improved in order to gain better performance. Therefore, a ray tracing

model utilizing line-of-sight information was developed. Rays are still traced, but the impact of ray tracing is considerably less in the modified model. The ray tracing takes place only at the beginning of the simulation, when rays sent from the base station are traced to all corners of the obstacles. Next, a line-of-sight (LOS) –polygon is generated to each corner. The LOS polygon indicates the area seen from the corners, denoting the area to which the rays diffract (Heiska and Kangas 1996). Finally, the actual coverage calculation includes "filling" of the LOS polygons by using the powers of signals at the corner as base power. The use of a more sophisticated ray tracing method had two benefits: the complexity of the simulator is reduced, and the method is more suitable for parallel processing due to the existence of LOS polygons.

Diffraction calculation is referred to as horizontal calculation. The name indicates that the rays are only traced on a certain horizontal level, i.e. height of the base station. Reflection calculation is part of horizontal calculation. Tests have indicated that the exclusive use of horizontal calculation does not provide adequate results. Therefore, vertical calculation is required to increase accuracy of the results. Vertical calculation consists of over-the-roof signal propagation. For each receiving point, a straight distance from the base station is calculated, and possible obstacles between the receiving point and the base station are observed. Based on the gathered information, the field strength to the receiving point is calculated.

PARALLELIZATION PROCESS

During the parallelization process, both vertical and horizontal calculations were modified to run in parallel. In the sequential simulator, vertical calculation consumed about 20% of the total execution time of the simulator. On the other hand, the execution of the diffraction calculation took up 75% of the total execution time. The remaining 5% was spent in initialization of the simulator and the parallel environment.

For both calculations, a work balancing algorithm was developed. The algorithms handled the equal distribution of work, which was necessary for the optimal utilization of processors, and the equal execution times of the individual processors. The work balancing algorithm implemented for each environment is discussed in further detail when the environments are presented.

There are several methods to convert a sequential program into a parallel program (Skillicorn and Talia 1998). On a general level, two main methods can be distinguished: threads and message passing. Threads provide a method to implement several instances of code running entities inside a single process. Each thread has its own memory for control structures, but all threads share resources allocated for the process. This makes the thread programming an attractive approach for shared memory machines. The parallelization of the GSM network simulator for the shared memory environment was implemented with POSIX threads (Butenhof 1997). The use of message passing in a shared memory environment is possible, but not advisable due to generated overhead and the existence of threads. In a distributed memory machine, message passing is required to enable communication between processors. Message passing was also used to convert the sequential simulator to the cluster of workstations.

A great number of messaging passing libraries are readily available (Dongarra 1995). Currently the most recognized message passing library is MPI (Message Passing Interface), which offers an interface for C and Fortran programmers to handle point-to-point and collective message passing (MPI Forum 1994). Nearly all computer vendors manufacturing workstations or multiprocessor machines have their own implementation of MPI. These specific versions of the MPI standard have been optimized for the particular systems in order to enable the utilization of system resources to their fullest.

Despite the chosen environment, the parallelization method always introduces overhead to computation. Overhead is the extra work required by the parallel application to communication, synchronize execution, protect critical areas, or create, maintain, and terminate parallel environments. For each environment, the amount and type of overhead varies. In a shared memory environment creating threads and protecting critical areas are the main contributors to overhead, whereas in a distributed memory, communication between the processors is by far the greatest source of overhead. The problems induced by overhead are discussed later when the environments are presented.

THE PARALLEL ENVIRONMENTS

Three distinctive environments were compared based on their speed, scalability, and cost-efficiency. In this subsection all three environments are presented. The special features, benefits, and drawbacks of each environment are detailed.

Shared memory environment

The shared memory environment comprised of a SMP (shared multiprocessor) machine with 8 processors. The processors were 435MHz Digital Alpha chips sharing a memory of 4 Gbytes. POSIX threads were used implementing the parallel version of the simulator for the environment. Although, MPI could have been used, threads generated less overhead. In the simulator, the threads were created at the beginning of computation and were not terminated until the program exited. This was essential, since an excessive amount of time is needed to create a thread. While using threads, it is required to provide a locking mechanism for shared variables, which can be accessed by all threads simultaneously. Concurrent access cannot be allowed in order to guarantee the correct execution of the program. Therefore, the POSIX thread library provides a number of locking mechanisms to protect shared variables. The mechanism used in the simulator is called a mutex (mutual exclusion) lock. The lock protects a critical section allowing only one thread to access the area at a time. However, the use of locks should be considered carefully since they pose the possibility of performance bottlenecks. Since a single thread executes the critical area at a time, the level of parallelism is diminished.

A dynamic work balancing algorithm was implemented due to the characteristics of the environment. In general, it can be said that dynamic work balancing requires communication between processors (Zaki et al. 1997). Therefore, the use of a dynamic algorithm is especially advisable in a shared memory environment, where the effect of communication is minimal. The work balancing algorithm created a pool consisting of work units to be processed. In vertical calculation, a unit represented a receiving point, and in horizontal calculation, a LOS polygon. Threads fetched a unit at a time from the pool. The use of the work pool in distributing work between processors automatically utilizes the processors to their fullest potential. In addition, in horizontal calculation, the LOS polygons were fetched from the pool in descending order, based on the amount of work in each polygon. The additional feature minimized, to some degree, the time at the end of parallel computation when threads were

synchronizing their execution; all threads had to complete their execution before the final results were produced.

Distributed memory environment

There are several similarities between a distributed memory machine and a cluster of workstations. The main reason of different approaches for the environments was the communication medium and the layouts of processors. In the distributed memory machine, the network connecting the processors (within a single machine) to each other is a high-speed network. On the other hand, in a cluster of workstations the interconnecting network is slower and the network is constructed of a number of computers.

The distributed memory machine used to run the GSM network simulator was Cray T3E. The T3E had 224 DEC Alpha chips with 128 Mbytes of memory each. The interconnecting network had a transfer bandwidth of 480 Mbytes/second. The processors formed a 3D torus, so that each processor had four neighbors (Cray Research 1996).

As mentioned earlier, in the distributed memory environment, the communication poses a great deal of overhead due to communication. To minimize the effect of communication, the implementation of the GSM network simulator was done so that communication took place before and after the parallel computation. The time spent in communication still contributed to overhead, but not as significantly as it might have done. MPI provided methods for collective communication, where with a single command more than two processors were involved in communication. The collective commands may be easy to use but they do not necessarily induce the best possible performance. In a distributed memory environment, as well as, in a cluster of workstations, the NFS (network file system) was used instead of the MPI routines to distribute the data between the processors. Each processor simply read the data from a file over NFS.

Despite the cost of communication in a distributed memory environment, a dynamic work balancing algorithm was developed. The concept of the algorithm was simtlar to the one implemented for the shared memory environment. The only exception was that each processor had a copy of all work units, whereas in a shared memory environment threads, used a single copy of the units stored in the memory of the process. A work pool was created and a master processor was dedicated to handle fetching of the work units from the pool. The master processor kept track of the units that had not been processed. Based on a request sent by a processor the master processor replied with an unit to process. Granted, the master-worker work balancing schema generated overhead in the form of communication during the parallel processing. Normally, the communication would not have been acceptable. However, SGI/Cray Research Inc. has developed an optimized message passing library for their systems to simulate a shared memory environment. The library is called SHMEM (Shared Memory), and it provides routines for the use of the distributed memory machine as a shared memory machine. With these routines processors can access other memories of other processors without any interference to the accessed processors. The SHMEM routines also automatically provide synchronization for the accessed data that prevents the data from being corrupted. In the simulator, a SHMEM routine was used to fetch the index to the next unit to process. As in the shared memory environment, applying a dynamic work balancing algorithm gave better performance through improved work balance, and processor utilization.

Cluster of workstations

Workstation clusters are becoming very popular among organization that are not willing to spend million of dollars to acquire a supercomputer. The term "cluster of workstations" however, can be misleading; a cluster can be formed from standard PCs (cluster of PCs, COP), or from a group of multiprocessor machines (cluster of SMPs) (Hwang and Xu 1997). The goal of cluster computing is to provide supercomputing capabilities by connecting a number of independent computers together. The Internet provides the ultimate environment for clusters. However, a more traditional cluster is formed from computers found in an office or at a university. The user interface of a cluster is relatively similar to a distributed memory machine. The main differences to a distributed memory machine are the interconnecting network, the heterogeneous environment, and the non-dedication of the system. The network connecting the workstations together is usually based on a TCP/IP protocol suite, and the network's bandwidth is not as high as in distributed memory machines. There are commercial solutions, such as Myrinet, for high-speed communication in clusters, which still are rather expensive to be used in an office environment (Hwang and Xu 1997). On the other hand, since a cluster is a group of computers, the computers are usually not identical. The heterogeneous environment poses dilemmas concerning work balancing due to the varied computing capacities of the workstations (Dongarra 1995; Zaki et al. 1997). The dedtcation of a system implies whether the system is dedicated entirely for parallel computing or not. A non-dedicated cluster allows the interactive usage of workstations during the parallel execution. Non-dedication leads to work balancing problems, since the use of a heavily loaded computer for parallel computing is not reasonable. A lot of research has been conducted concerning work balancing in heterogeneous environments (Hui and Chanson 1997; Schlahenhaft et al. 1995).

The cluster where the test runs of the GSM network simulator were conducted consisted of 16 dual Pentium workstations running the Linux operating system (CSC 1999). The workstations were interconnected with a Fast-Ethernet switch (100 Mbits/s). MPICH (MPI Chameleon) was the implementation of the MPI message passing interface in the cluster. MPICH is, perhaps, the most commonly used implementation of the MPI due to its good performance-portability ratio (Gropp et al. 1996). Since the implementation of the MPICH did not support the use of a shared memory as a communication medium between the processors within a workstation, all communication took place via sockets.

The implementation of the GSM network simulator for the distributed memory machine was used as a basis for the implementation in the cluster of workstations. NFS was also utilized to relay the data to all processors in a cluster. However, due to the high cost of communication in a cluster of workstations, the work balancing algorithm in horizontal calculation was modified. A static work balancing was implemented to minimize communication. On the other hand, a static work balancing algorithm is rarely capable of creating equal work balance; the trade-off between optimal work balance and communication had to be made. Firstly, the work balancing algorithm sorted LOS polygons based on the work required by each polygon. From the sorted pool of polygons, processors fetched a polygon to process, starting with the polygons requiring the most work. Since the algorithm was static, each processor processed a fixed number of polygons, based on the processor id number.

Figure 6 illustrates how the distribution of polygons to the processors is done. The number of processors is four (id =

0..3). Since each processor has a fixed number of polygons to process, the optimal work balancing is next to impossible to attain. To sort the polygons and to use interval fetching of polygons the most optimal static work balancing algorithm can be implemented.

Polygons	1	2	3	4	5	6	7	8	9	10	11
Processor	0	1	2	3	0	1	2	3	4	0	1

Figure 1. Distribution of the polygons to the processors.

SIMULATION EXPERIMENTS AND RESULTS

The same simulation was performed on the three environments. The tests were run from one processor to the maximum number of processors in a system., except in Cray T3E where the maximum number of processors was 32. The execution times varied considerably from one environment to another. For instance, the same simulator run with only one processor produced the following execution times:

- Shared memory environment, 255.75 seconds
- Distributed memory environment, 736.58 seconds
- Cluster of workstations, 1188.65 seconds

The differences can be explained with the variation of processor speeds, and memory hierarchies. Therefore, speedups were calculated based on the sequential execution time of the GSM network simulator. This made it possible to compare the achieved results. The comparison based on speedups had a single drawback; the speedup does not take the deviation of the execution times of different environments into consideration. Thus, the usage of speedups to compare environments is not necessarily the best method with respect to the execution time. The evaluation of speedups deals with the scalability capabilities of systems, as well as, the issues relating cost-efficiency.

As a test case, coverage generated by a single base station in an urban environment was calculated. The map over which the coverage was calculated consisted of the city of Helsinki. The calculation area was 1640 x 1470 meters (2.4 km^2), and the size of a receiving point was 4 x 4 meters. The achieved speedups are shown in Figure 2. Since threads were used in the implementation for the shared memory environment, the speedups are shown against the number of threads instead of processors. It is possible for the number of threds to exceed the number of processors, as seen in Figure 2. In some cases, the creation of extra threads can provide increased performance (Butenhof 1997). However, in this case, no performance gain was achieved.

The results indicated that the best speedup was achieved with the PC cluster (13.3). However, the performance of Cray T3E did not differ significantly. The results from Cray T3E are considerably more stable than the cluster environment. This is the result of the communication network of the PC cluster. The cluster was a collection of workstations in a TCP/IP network, which was part of an office-wide network. Each workstation also ran other programs during the parallel execution of the simulator. The dedicated use of Cray T3E, and the high-speed interconnection network provided a more stable platform for parallel execution.

The shared memory machine had only 8 processors, which limited the achieved results. Even if the comparison had been done within the range of 1 to 8 processors, the shared memory machine performed the worst. This can be considered as an unlikely event, since normally the amount of overhead created in the shared memory environment is minimal. However, in this case there was no need for communication during the calculation, which eliminated the advantage of the shared memory. On the other hand, since threads shared the memory, synchronization methods were implemented generating overhead. Synchronization with overhead created by manipulation (creation, scheduling, destruction) of threads, caused the shared memory environment to perform more poorly than the two distributed environments. However, the shared memory environment was the only environment where the system was not even partly dedicated to running the GSM network simulator.

The implementation for Cray T3E used specific communication routines optimized for the Cray MPP environments. Without the use of the SHMEM library calls the performance of Cray T3E would have been worse. For the cluster environment, only the basic MPI library routines were utilized. Therefore, to conclude, the cluster of workstation can be said to perform the best out of the three environments presented in this paper. The cost of creating and maintaining a cluster makes it an especially

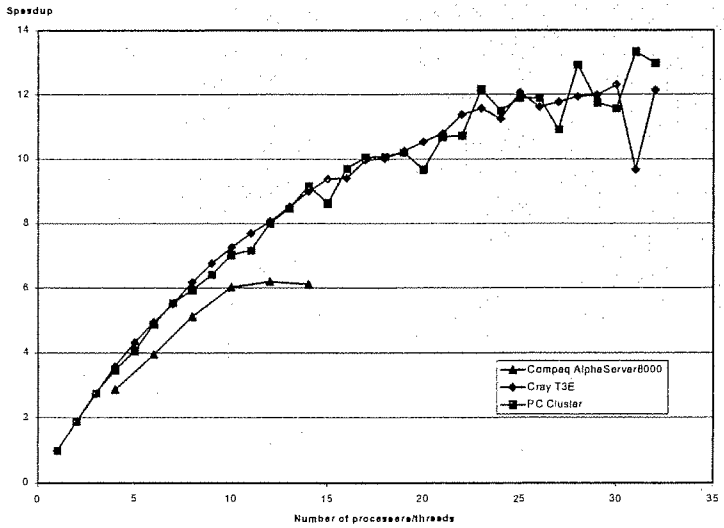


Figure 2. Speedup of the simulation

lucrative choice for a parallel environment. Similar speedups were achieved with a multimillion dollar Cray T3E supercomputer, and with a cluster of workstations worth about $50,000. Since the cluster does not need to be dedicated to parallel computing, the workstations forming the cluster can be machines in an office or at a university. The cluster enables flexible configuration, and the efficient utilization of resources; the machines can be used to run sequential and parallel applications simultaneously. The size of the cluster can be easily adjusted by increasing the number of computers participating in the calculation, whereas in an MPP system new processors have to be bought with other related hardware (e.g. motherboards, and memory).

It should be noted, that Figure indicated the total speedups of the execution times. The two parallelized parts produced significantly better speedups. For example, in Cray T3E, the maximum speedup for vertical calculation was 12.6. However, the amount of work required for vertical calculation restricted the achievable speedup. The execution time of vertical calculation was reduced from 20.2 seconds to 1.6 seconds. Horizontal calculation had much more work to distribute between processors. Therefore, the achieved results were noticeably better than those from vertical calculation. The maximum speedup of 22.8 was obtained with the use of 32 processors; the parallel execution time dropped from 689.3 seconds to 30.24 seconds. Similar behavior was detected in the PC cluster.

The study of the performance of a cluster environment in this paper has shown that the execution times can be reduced significantly. However, the problem that remains in a cluster environment is the effect of communication to the actual calculation. This dilemma did not concern the GSM network simulator, due to the structure of the simulator. Nevertheless, the results showed that minimizing the need for communication makes the cluster of workstations an environment worth considering when the need for parallel computing arises.

CONCLUSIONS

The purpose of this paper was to present three different parallel environments with their special features, and compare performances of the environments. A GSM network simulator was parallelized for each environment by using the optimal programming methods offered by the systems. For a shared memory environment threads were utilized to run the simulator on multiple processors. In a distributed memory environment, MPI and SHMEM library calls were utilized. The SHMEM library calls were special routines created for Cray's MPP environments. In the cluster environment, the implementation was done by using only the MPI library calls.

Work balancing algorithms were developed for each environment to take advantage of their characteristics. The main goal in a shared memory environment was to minimize the use of synchronization primitives, whereas in a distributed memory environment and a cluster of workstations, efforts were made to minimize communication. In general, the significance of a work balancing algorithm cannot be overlooked in parallel computing. An incorrect work balancing algorithm can crucially inhibit performance of the parallel application.

The test runs of the simulator were run with similar parameters in all environments. The achieved results showed that the shared memory environment was inferior to the distributed and cluster environments. There was not a significant difference in performances of the distributed and cluster environments. The comparisons were made based on the achieved speedups. The comparisons gave an indication of how well the environments performed when more processors were added.

To conclude, the comparison presented in this paper showed that the cluster of workstations is definitely an option worth considering for parallel computation purposes. First of all, the cost of creating a cluster is low. In addition, the workstations that form the cluster can be in every day use by staff and students at a university. The parallel jobs can be run concurrently with the jobs created by interactive users. Whereas a distributed memory machine usually has a set of similar processors (speed and memory), a cluster of workstation can consist of machines with different parameters: number of processors, processor speed, amount of memory, architecture, and operating system.

References

Butenhof, D.R. 1997. *Programming with POSIX Threads*. Addison-Wesley, ISBN 0-201-63392-2.

Center for Scientific Computing (CSC). 1999. http://www.csc.fi/metacomputer/pckluster

Cray Research. 1996. Cray T3E Applications Programming (TR-T3EAPPL (D)).

Dongarra, J.J. 1995. "Heterogeneous Network-Based Concurrent Computing Systems." In *High Performance Computing: Technology, Methods and Applications*. Elsevier Science B.V., ISBN 0-444-82163-5, pp. 5-16.

Gropp, W.; E. Lusk; et. al. 1996. "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard." Technical Report ANL/MCS-P567-0296, Argonne National Laboratory. (July).

Hata, M. 1980. "Empirical Formula for Propagation Loss in Land Mobile Radio Services." *IEEE Transactions on Vehicular Technology*, vol. VT-29, no. 3.

Heiska, K. and A. Kangas. 1996. "Microcell Propagation Model for Network Planning." In *Proceedings of IEEE PIMRC 96*. (Taipei, Taiwan).

Hui, C-C. and S.T. Chanson. 1997. "Theoretical Analysis of the Heterogeneous Dynamic Load-Balancing Problem Using a Hydrodynamic Approach." *Journal of Parallel and Distributed Computing*, vol. 43.

Hwang, K. and Z. Xu. 1997. *Scalable Parallel Computing*. McGraw-Hill, ISBN 0-0703-1798-4.

Message Passing Interface Forum (MPI Forum). 1994. "MPI: A Message Passing Interface Standard." *International Journal of Supercomputer Applications*, 8 (3/4).

Rappaport, T.S.; R. Muhamed; et. al. 1996. "Propagation Models." In *The Mobile Communication Handbook*, Chapter 22. CRC Press, Inc. ISBN 0-8493-8573-3.

Schlagenhaft, R.; M. Ruhwandl; et. al. 1995. "Dynamic Load Balancing of a Multi-Cluster Simulator on a Network of Workstations." In *Proceedings of Parallel and Distributed Simulation* (Lake Placid, USA).

Sipilä K. and K. Heiska. 1996. "Can ray tracing be used as a fading generator in simulating micro cellular mobile radio system?" In *Proceedings of the 8th International Conference on Wireless Communications* (Calgary, Canada).

Skillicorn, D.B. and D. Talia. 1998. "Models and Languages for Parallel Computation." *ACM Computing Surveys*, vol. 30, no. 2.

Wilkinson, B. and M. Allen. 1999. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, ISBN 0-13671710-1.

Zaki, M.J.; W. Li; et. al. 1997. "Customizing Dynamic Load Balancing for a Network of Workstations." *Journal of Parallel and Distributed Computing*, vol. 43.

Publication 2

Huttunen P., Ikonen J., and Porras J.: The Impact of Communication in Distributed Simulation. In Proceedings of European Simulation Symposium, Marseille, France, October 18-20, 2001, pp. 111-115.

THE IMPACT OF COMMUNICATION IN DISTRIBUTED SIMULATION

Pentti Huttunen
Jouni Ikonen
Jari Porras
Lappeenranta University of Technology
Department of Telecommunication
P.O. Box 20, FIN-53851 Lappeenranta
Finland
{Pentti.Huttunen, Jouni.Ikonen, Jari.Porras}@lut.fi

ABSTRACT

The ever-increasing computation requirements of present day simulators have prompted a need for distributed simulation. There are two main improvements in the distributed simulation over the sequential simulation: the execution times of the simulation runs are significantly reduced, and bigger simulation problems can be solved. Unfortunately, the distribute simulation does not come without drawbacks. As the name implies, the simulation is divided into independent parts and distributed over a number of processors. The distribution involves communication among processors. From the parallel computing point of view the communication is additional work that does not exist in the sequential simulation. Thus, the communication is considered overhead. The communication overhead can have a significant impact on the total execution time of the distributed simulation. The research presented in this paper studies the effect of communication in regard to the performance of the distributed simulation.

1. INTRODUCTION

Due to the complexity of present day simulators, the usage of distributed methods to run simulators has increased drastically. The distributed simulation has been proven to boost the performance, and to allow the execution of bigger and more complex simulations. There are a number of dilemmas faced by the simulation expert while implementing a distributed simulator or converting a sequential simulator to be run in parallel. The two potential problem areas are load balancing and communication. Load balancing deals with dividing the simulation into independent parts and assigning the parts to the processors for execution. A large research community is studying load balancing. Unfortunately, there is still not a single way of handling load balancing for all simulation problems in all distributed systems. Thus, load balancing will remain an intensive area of research for a number of years. The second problem is the communication. Once the load balancing algorithm has calculated the work distribution, the information has to be communicated to all the processors. Furthermore, the processors may need to transfer data among themselves during the computation phase. Whereas load balancing usually takes place prior to the actual parallel computation, the processors can resort to communication at any point of the execution. Therefore, the communication has a profound effect on the execution of the simulator. The purpose of the study presented in this paper is to quantify the

contribution of communication and describe methods to reduce the negative impact of communication to distributed simulators.

The structure of the paper is as follows. The second chapter discusses distributed simulation. It details issues unique to parallel simulation such as load balancing, synchronization and communication. The third chapter is devoted to communication. The effect of communication is discussed in conjunction with solutions to overcome the dilemmas cause by the communication. The fourth chapter presents the performance results of a TCP/IP network and the Message Passing Interface (MPI) library. The first part of the tests measures the performance of the 100 Mbps Ethernet network with the most common protocol TCP/IP. The second part of the measurements describes the performance of a message passing interface (MPI) that has been implemented on top of the TCP/IP protocol. Finally, the last chapter concludes the paper.

2. DISTRIBUTED SIMULATION

Simulation is considered distributed if parts of it are executed concurrently on a number of processors. The actual implementation of the distributed simulator is greatly dependent on the phenomenon to be simulated and on the simulation expert. There are a vast number of tools to assist the simulation expert in implementing the sequential simulator. Unfortunately, the same is not true for the distributed simulator. Most of the distributed simulators are nowadays implemented with traditional programming languages such as C, C++, and Fortran. The simulation expert is responsible of coding the simulation along with all the components required by the distributed execution. The main components are load balancing, synchronization, and communication.

Regardless of the significant effect of load balancing in distributed simulation, load balancing is not discussed in detail. The load balancing is a very complex and important task to be conducted prior or during the parallel simulation. The interested reader is referred to (Wilson and Nicol 1996; Wilson and Shen 1998) for more information about load balancing in distributed simulation. More generic information about load balancing is found in (Ezzat 1986; Hamdi and Lee 1995; Ros and Yao 1991).

Synchronization deals with situations where the execution of the processors has to be coordinated. For example, a processor that calculates the final results based on the sub-results of all other processors has to wait until the processors have produced the sub-results prior to computing the final results. In an optimally balanced simulator, the synchronization overhead is minimal. If all processors arrive at a synchronization point at the same time, the execution of all the processor continues immediately. Unfortunately, the amount of work on the processors is not usually equal, which contributes to the synchronization overhead; a processor is not allowed to continue its execution until all the processors have entered the synchronization point. Furthermore, synchronization requires communication. The processors have to communicate with each other to indicate when they enter the synchronization point and when they can continue their execution. The effect of the communication caused by the synchronization is usually ignored. However, a synchronization routine can

create a great deal of small messages that are relatively expensive to transmit. The *test case and results* chapter discusses more about the cost of transmitting small messages.

Finally, the last component is the communication generated by the load balancing algorithm and the execution of the simulation. With a proper implementation of the load balancing algorithm the amount of communication can be minimized. This is especially beneficial in distributed systems where the communication network is slow. In practice, there are two ways to minimize the communication. First, the same load balancing algorithm is executed on all the processors allowing each processor to know the global load balance. Second, the load balancing algorithm is executed on a single processor, which is the common way with a high-speed interconnecting network. However, in the second scenario the workload is not distributed over the network by broadcasting the data from one processor to all the others. The distribution is done through a third-party application or device. For example, the processor that calculated the load balance sends only the indexes for the workload, and all the processors read the workload from a hard drive. This requires that the processors have access to the workload data. Network File System (NFS) can provide the access. Although, the processors resort to the same interconnecting network to access the shared hard drive, the access times are reduced. The usage of NFS is more optimized than usage of a message passing library to communicate between processors on an application level. The following chapter discussed in further detail about the communication in distributed simulation.

3. COMMUNICATION

There is, practically, always a need for communication in distributed simulation. Obviously, the communication should not dominate the execution of the simulation whereas it should be a minor contributor to the total execution time. However, unless the communication is designed and optimized properly its effect can be substantial diminishing the gain achieved by executing the simulation in parallel.

If the simulator is implemented with Fortran or C, there are a handful of message passing libraries that are widely used. Perhaps, the most common message passing library is called Message Passing Interface (MPI) (MPI 1995; MPI 1997). It offers the simulation expert a relatively comprehensive set of function calls to handle point-to-point and collective communication. The main purpose of MPI is to make the communication easy to implement but still be efficient from the performance point of view. MPI's main competitor is the Parallel Virtual Machine (PVM) library (Sunderam 1990). PVM is also a collection of function calls for Fortran and C. However, PVM is more versatile with respect to the interface and the usage. In practice, PVM has compromised the performance with a greater number of features, whereas MPI provides optimal performance with a limited set of communication functions. Naturally, the simulation expert has the option of implementing the simulator merely deploying the functionalities provided by the programming languages such as sockets and pipes (Stevens 1990). However, the previously mentioned message passing libraries utilize sockets but hide the internal implementation from the user. In general, the simulation expert should not try to implement his own communication library since such libraries already exist and have been extensively optimized and tested. Although, the experiment results shown in the

4.2. Bandwidth

Bandwidth defines the throughput of the transmission. It can vary considerably depending on a number of characteristics of the network, such as the hardware, the network usage, and the protocol. The bandwidth measurements were conducted in an "empty" cluster; no other user processes were present, and the only network messages were generated by the necessary daemons running in the systems.

Figure 1 shows the measured communication times for message sizes from 1 byte to 100 Mbytes. The communication times for small messages (1 to 1000 bytes) are below 0.1 milliseconds for TCP. However, for larger than 1000-byte messages, the communication time increases exponentially. The performance of MPI is nearly identical with TCP. The major difference from the performance of TCP can been seen with smaller messages (size less than 4 Kbytes. Unfortunately, the smaller messages are more commonly used than larger messages. For example, synchronization messages and load balancing information are sent with messages that sizes are far less than 4 Kbytes. MPI suffers from a high overhead with small message sizes due to the encapsulation. Regardless of the performance difference in small messages, the deployment of the MPI library instead of the user's own implementation of a message passing library is justified.

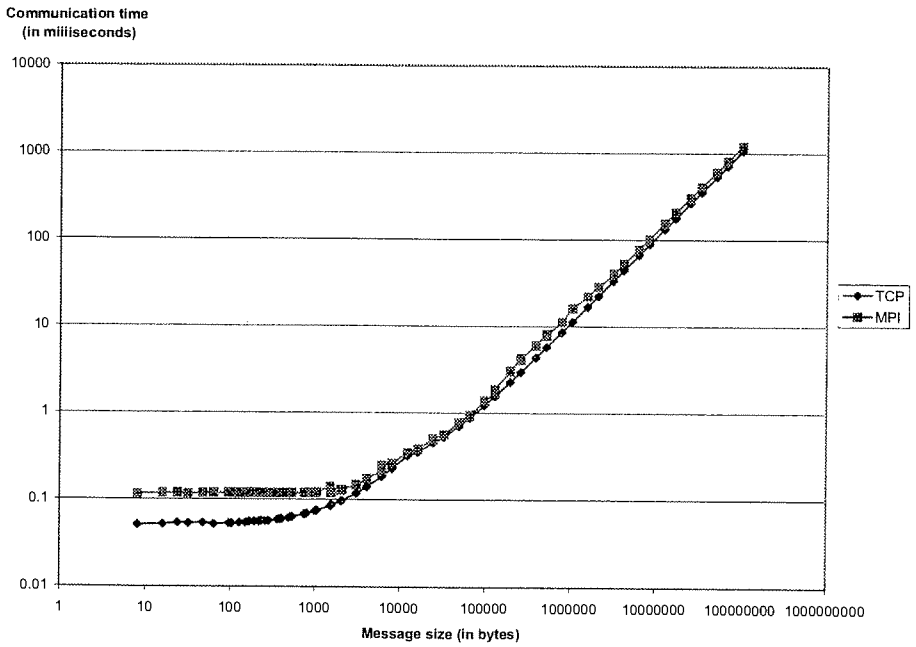


Figure 1. Measured communication times with TCP and MPI.

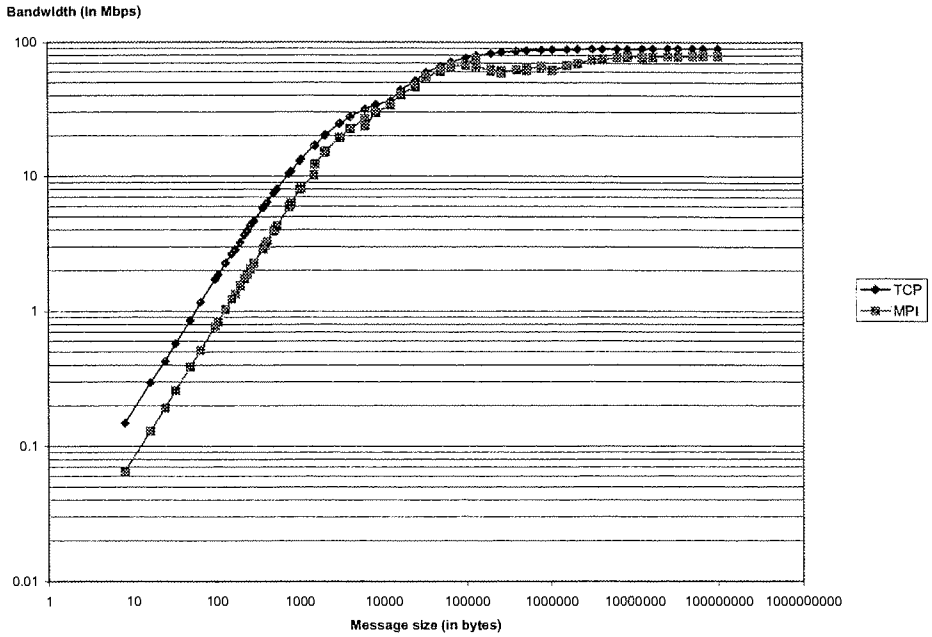


Figure 2. Measured bandwidths for TCP and MPI.

Figure 2 depicts the measured bandwidth of the network as a function of a message size. Obviously the smallest messages are no capable of utilizing the available bandwidth of the network due to the short transmission time. As the message size increases so does the bandwidth usage. In TCP the bandwidth begins to saturate when the message size reaches 100 Kbytes. The TCP protocol is capable of utilizing 90 % of the theoretical maximal transfer rate of 100 Mbps. Thus, the maximal transfer rate achieved was 89.76 Mbps (11.22 Mbytes/s). The behavior of MPI is not equally good. For small message the bandwidth utilization follows TCP with a certain reduction though. However, with messages over 132 Kbytes, the bandwidth utilization degrades. There is a slump in the curve until the message size of 3 Mbytes is reached. The measurements were run multiple times to rule out any possibilities to measurement errors. Furthermore, a very interesting point is to notice that the curve actually saturates after all. The reason why MPI behaves such as seen here could not be explained. Overall, MPI performance from the bandwidth point of view is still adequate. With very large messages the bandwidth saturates at 79.49 Mbps (9.94 Mbytes/s) that is approximately 88.6 % of the maximum bandwidth measured with TCP. This reduction is a result of the overhead incurred by MPI due to the encapsulation.

In order to estimate the impact of the communication on the execution of a distributed simulator, an execution time of a division operation was measured. In general, the division operation, especially one that cannot be computed with bit shifts, is considered the most expensive operation among the arithmetical operations. The measurements showed that a single processor on a workstation is capable of executing 114 311 floating point division operations in a second. Thus, during a transmission of, say, a 32-Kbyte message a single processor could perform 500 divisions operations.

The number does not sound that significant, although the actual simulation could have made progress substantially, if the processor had been available. Figure X depicts the number of divisions operations that could have been processed during the communication in TCP and MPI.

The latency plays a significant role in the communication times of small messages. Figures 3 shows that the TCP and MPI curves have similar shapes. However, MPI experience a considerable larger overhead with messages smaller than 1 Kbyte experience similar communication times. This is due to the fact that the latency counts over 79 % of the total communication time in MPI, whereas in TCP the corresponding percentage is 23. When the message size increases the contribution of the latency becomes insignificant. However, smaller messages are widely used in simulators for synchronization and load balancing. In order to optimize the performance of a simulator the number of messages should be minimized meanwhile maximizing the size of the messages. This can be achieved by compounding a number of small messages into a single large message.

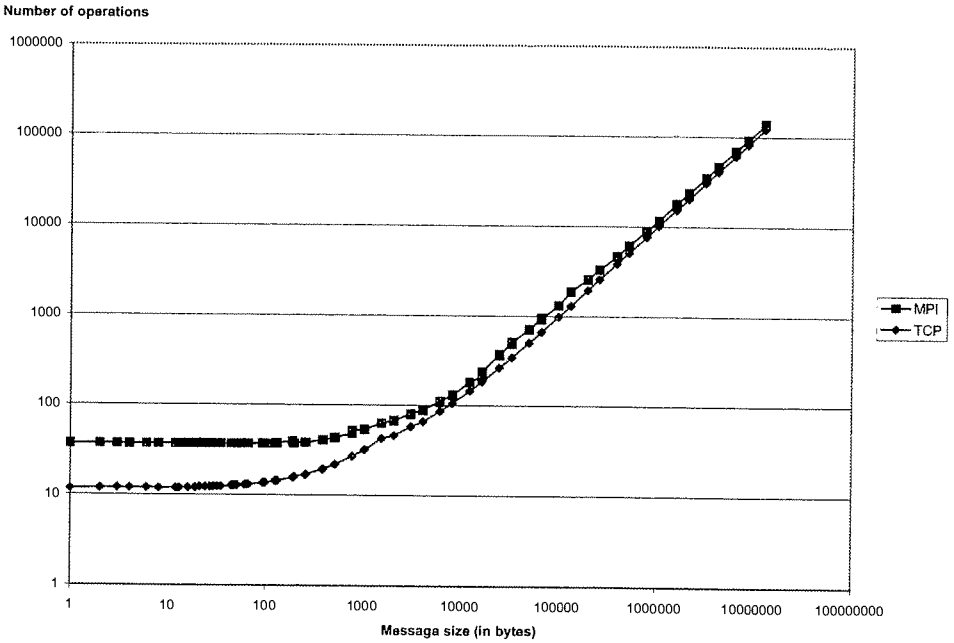


Figure 3. Number of floating point division operations during the communication.

If the achieved results are applied to a distributed GSM network simulator (Huttunen 2000), the following observations are made: It would take 8.137 milliseconds to transmit an average size message (65000 bytes). During that period of time 930 floating point operations could have been computed. This number shows the significant overhead created by the communication. Furthermore, the number of operations (930) are that are performed is only for a single message. In practice, during the course of the simulation, a multitude of messages (smaller and considerably larger) are sent and received by the processors.

It is possible for the communication and the computation to overlap to a certain extent. With a proper MPI functions the overlap can be explicitly implemented. However, to maximize the overlap the processing element has to have two processors; one for the actual computation, and one for the communication. The communication processor is dedicated to transmit and receive messages. The processors can be similar with respect to their architecture and performance. Since the sole purpose of the communication processor is to provide access to the network, it is more likely to be optimized for handling data transmission. On the other hand, if not always is it feasible to process data while a communication operation takes place. For example, if the processor needs the data being transmitted in order to proceed with the execution of the simulation, the overlap is not possible.

5. CONCLUSIONS

Message passing times of a parallel simulation utilizing TCP and MPI for communication were measured on a cluster of workstations with a 100 Mbps Ethernet network. The cluster environment was homogeneous; each workstation had two Pentium III 800 MHz processors and 2 Gbytes of memory. The purpose of the study was to quantify the effect of communication to a parallel simulation that needs to transfer data among processors before, during, and after the parallel computation phase.

The measurements indicated that the latencies in TCP and MPI are major overhead in small messages. Especially, the MPI latency is significant. However, the latency becomes insignificant as the message size grows. The communication time was observed to grow linearly in TCP in accordance with the message size. MPI incurs additional overhead due to the encapsulation of the message, and, therefore, fails short of the performance of TCP. The bandwidth measurements reflect the same problem with small messages as was seen in the latency measurements; TCP outperforms MPI due to the additional overhead of MPI. The difference in the maximum bandwidth is approximately 15 %, which is acceptable in a 100 Mbps network. The peak bandwidth of MPI was measured at 79.9 Mbps. Furthermore, programming with the MPI library is considerably easier than implementing the communication with TCP (sockets). Therefore, a compromise between the performance and ease-of-use has to be made.

The study clearly illustrated that the communication has a major impact on the overall performance of a parallel simulator. Thus, during the design phase of the parallel simulator it is essential to consider the number and the size of the messages sent in the course of the execution of the simulator. By carefully determining the optimal message size, the impact of latency is minimized while throughput is maximized. However, it is important to realize that the optimal message size varies from network to network as well as from protocol to protocol.

REFERENCES

Ezzat, A.K. 1986. "Load Balancing in NEST: A Network of Workstations." in proceedings of ACM/IEEE-CS Fall Joint Computer Conference: 1138-1149.

Gropp W.; E. Lusk; N. Doss; and A. Skjellum. 1996. "A high-performance, portable implementation of the (MPI) message passing interface standard." *Parallel Computing*, Vol. 22, No. 6 (Sept), 789-828.

Hamdi, M.; C-K Lee. 1995. "Dynamic Load Balancing of Data Parallel Applications on a Distributed Network". In *Proceedings of ACM International conference on Supercomputing*. 170-179.

Huttunen, P.; J. Porras; J. Ikonen and K. Sipilä. 1998. "Parallelization of Propagation Model Simulation". In *Proceedings of European Simulation Symposium* (Nottingham, England). SCS Europe.

MPI Forum. 1995. "MPI-1.1 Standard."

MPI Forum. 1997. "MPI-2 Standard."

AMES Laboratory. 1995. Network Protocol Independent Performance Evaluator (NetPIPE), Iowa State University, http://www.scl.ameslab.gov/Projects/Netpipe/.

Pacheco, P.S. 1997. *Parallel Programming with MPI*. Morgan Kaufmann, San Francisco, CA.

Ross, K.W.; D.D. Yao. 1991. "Optimal Load Balancing and Scheduling in a Distributed Computer System" *Journal of the Association for Computing Machinery*, Vol. 38, No. 3, 676-690.

Stevens, W.R. 1990. *UNIX Network Programming*. Prentice-Hall, Canada.

Sunderam, V.S. 1990. "PVM: A Framework for Parallel Distributed Computing." *Concurrency: Practice and Experience*, Vol. 2, No. 4, 315-339.

Wilson, L.F.; D.M. Nicol. 1996. "Experiments in Automated Load Balancing". In *Proceedings of 10th Workshop of Parallel and Distributed Simulation*. 4-11.

Wilson, L.F.; W. Shen. 1998. "Experiments in Load Migration and Dynamic Load Balancing in SPEEDES". In *Proceedings of Winter Simulation Conference*. 483-490.

Publication 3

Huttunen P., Porras J., and Ikonen J.: A Study of Threads and MPI libraries for Implementing Parallel Simulation. In Proceedings of European Simulation Symposium, Hamburg, Germany, September 28-30, 2000, pp. 96-102.

A Study of Threads and MPI libraries for Implementing Parallel Simulation

Pentti Huttunen, Jari Porras, and Jouni Ikonen
Lappeenranta University of Technology
P.O. Box 20, FIN-53851 Lappeenranta, Finland
{ Pentti.Huttunen, Jari.Porras, Jouni.Ikonen }@lut.fi

KEYWORDS
MPI, threads, parallel simulation

ABSTRACT

This paper covers the two most popular methods for implementing parallel code: Threads and Message Passing Interface (MPI). Both methods are discussed in detail to provide information about the implementation issues of the methods. An indepth look is taken into the parallelization libraries that are widely used among programmers. The paper also describes, how to write parallel code by using these methods. In addition, two characteristics of parallel computing, synchronization and load balancing, are explored. Finally, a performance study of both methods is presented.

INTRODUCTION

The paper introduces two widely utilized libraries that can be used to parallelize existing applications, as well as, implement new parallel code. With the knowledge acquired from this paper, simulation experts and software developers responsible for implementing simulators should be able to determine whether their simulation system is suitable for parallel execution. In addition, the paper describes how to use both libraries.

The need for parallel simulation, as well as parallel computing in general, has increased drastically during the last two decades; more complex models and programs have been implemented. To meet the need for increased computation capabilities, the use of multiple processors has been proposed as a possible solution. In theory, the utilization of multiple processors seems straightforward. However, before an application can take advantage of multiple processors, the structure of the application and the code have to be modified. Depending on the current structure of the application, the issues concerning structure may not be as important as the modification of the code.

On the other hand, the utilization of multiple processors can be understood in more than one way. The simplest method to utilize a multiprocessor computer is to run a copy of the application on all processors. In this case, there is no need for the reconstruction or modification of the code. However, the approach does not decrease the execution time of an application, which is one of the main goals of parallel computing. Therefore, running several copies of an application does not constitute parallel computing, and is not discussed any further in this paper.

To decrease the execution time of an application, the application has to be divided into parts that are executed on different processors. The parts of the application must meet certain requirements. First of all, the parts need to be independent, meaning that there are no dependencies between parts. In real life, this requirement cannot be fulfilled completely. Therefore, parallel computing usually needs a synchronization method that controls the interactions between dependent parts. Synchronization is a key element of parallel computing, which has both positive and negative effects on the parallel execution of an application. Since the effect of synchronization is significant, issues concerning synchronization are considered later in this paper. The other requirement states that the sizes of the independent parts should be equal. As with the first requirement, it is practically impossible to create parts that are exactly equal in size. However, if the processors in the computer are not identical, the sizes of the parts should be adjusted to compensate for the differences in processors; a bigger part should be created for the fastest processor. The creation of parts to match the processing capabilities of processors is called load balancing. Load balancing is another crucial element in parallel computing, and it is discussed in detail later.

Having defined the requirements for parallel computing, the software engineer's first decision concerns the selection of the parallel environment. In essence, there are two parallel environments, based on the memory architecture, into which most of the present day multiprocessor machines fall. A shared memory machine has a single memory, which is accessible by all processors via a high-speed bus. Since each processor has access to the whole memory, the data exchange (communication) between processors can be carried out with the use of the memory. A distributed memory machine is an environment where each processor has its own memory. The processors are not connected to each other with a bus, but rather with a high-speed network. If processors need to communicate, they use the network to send and to receive data. However, the topology of the interconnection network depends on the machine.

Due to the diversities between the two environments, the programming models are also different. For implementation into a shared memory environment, threads are used to run the parts of the application on different processors. The implementation for the distributed memory machine, on the other hand, requires the use of message passing interface. Currently there are several message passing interfaces available commercially. However, there are also numerous interfaces that can be downloaded from the

Internet without charge. The Message Passing Interface (MPI) is one of the most popular and easy to use implementations of the message passing interfaces. The advantage of the MPI over threads is that the MPI can be used in shared memory machines as well.

The next chapters describe the two programming models, give insight into how to use the models, and compare their characteristics, especially in regards to performance.

THREADS

The basic entity in an operating system is called a process. Threads are instances of code that are created and run inside the process. Figure 1 illustrates the relationship between processes and threads. Each process, regardless of how many processors are in the computer, has one thread. This thread is created by the operating system when the process is started. However, the software developer can create more threads. The threads share all the resources allocated for the process; but, each thread has its own data structure that contains information about the particular threads, such as thread id, program counter, and local variables. Each thread is assigned a part of code that it executes. Several threads can execute the same code with similar or different parameters.

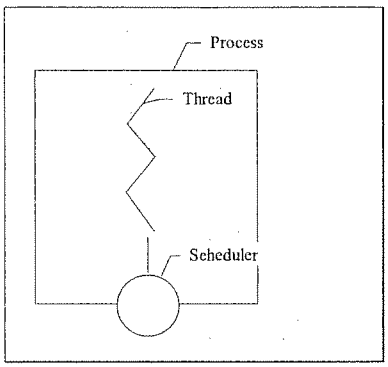


Figure 1. The structure of a process.

A thread is the entity that actually executes the code. The thread needs to be scheduled to a processor before it can be executed. In a process there is a scheduler that dynamically schedules threads to processors. Since each process can have several threads, the scheduler is of great importance. The responsibility of the scheduler is to allocate time for threads on processors according to the priorities of the threads, but it is also possible to bind a certain thread to a processor, meaning that the processor is dedicated for the use of a single thread. The bound thread then has a direct access to a processor without interference from the scheduler. A thread running a high priority real-time application should be bound to a processor to guarantee the best possible performance; Figure 2 illustrates this schema. As Figure 2 indicates, the scheduled and bound threads do not have direct access to processors. Each thread is actually scheduled or bound to a kernel thread that is executed on a processor. The operating system that supports threads creates and maintains the kernels threads.

It should be noted that the amount of threads inside a process is not limited to the number of available processors. It is possible to create multiple threads within a process in a uniprocessor machine. Before multiprocessor computers became commonly available and affordable, threads were used to provide concurrency in a machine with one processor. Most of the UNIX operating systems do multitasking by utilizing threads, which enables multiple applications to be run concurrently, although, not simultaneously. If a thread makes a system call that halts the execution of the application until the system call returns, another thread can be run in the meantime. Thus, if a thread is blocked by a system call or some comparable action, the thread is retrieved from the processor, and another thread can start execution. With this implementation, the performance of a uniprocessor computer increases significantly.

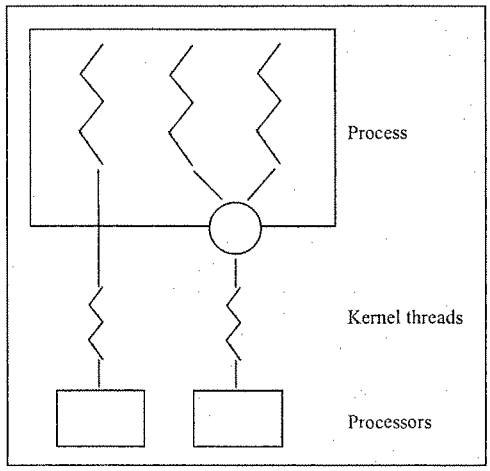


Figure 2. Mapping of threads to kernel threads and processors.

For parallel computing purposes, the number of threads should be near the amount of processors in a particular computer. Under normal circumstances, the best performance is achieved with a fewer number of threads than processors, because of other applications running on the computer, and the overhead created by threads. Nevertheless, the rule of thumb is to create no more threads than there are processors.

Thread Programming Model

At the present time, there are two thread libraries that are readily available. The first and more widely used thread programming model is called POSIX threads (pthreads). Pthreads are part of a group of standards defined in the Portable Operating System Interface (POSIX) by IEEE. All computer and operating system vendors have included support for POSIX threads in their systems. The other thread programming model is called Solaris threads. The Solaris threads work only under Solaris or Sun operating systems. Since the use of the Solaris threads is limited to Sun Microsystems operating systems, the thread package is more optimized and efficient than POSIX threads. However, due to the popularity and portability of the POSIX

threads programming model, this paper concentrates solely on POSIX threads.

The utilization of threads in making an application run in parallel requires at minimum one command, which then creates the threads. At the creation, each thread is assigned an identification number, parameters, a function to execute, and an input parameter for the function. The identification number (thread id) is a unique number assigned to each thread, and used mainly during the run-time to distinguish threads from each other. The identification number is stored in the variable specified in *pthread_create()* command, and is read only after that. The parameters define the characteristics, such as scheduling policy, of the created thread. In most cases, the default parameters (NULL) can be used. Since threads exist inside a process they must be assigned a portion of the application to execute. Usually, the threads execute the same function with different input parameters. It is the responsibility of a load balancing algorithm to divide the workload between the threads during their execution. The input parameters define the parameters for the function that the thread executes. Figure 3 shows an example of how to create a thread.

```
#include <pthread.h>

void *function( void parameter );


pthread_t thread_id;
int parameter;

pthread_create( &thread_id, NULL, function,
parameter );
```

Figure 3. Example of creating a thread.

As shown in Figure 3, all POSIX thread related commands are stored in a header file called pthread.h. It should also be noted that the function the thread executes needs to have a return type of a pointer to void. The input parameter for the function has to be a type of void as well. The thread creation command allows only one input parameter to be passed to the function. Therefore, if more than one parameter is required, a structure has to be made that contains all the input parameters. Then, the structure can be passed as one parameter to the function.

Table 1 lists the most useful commands in the POSIX thread programming model. These commands provide information about the threads, and assist in controlling the execution of threads. A more comprehensive list of commands and information about threads can be found in (Butenhof 1997; Lewis and Berg 1998).

Table 1. POSIX thread commands.

pthread_create()	Creates a thread.
pthread_attr_init()	Initializes attribute structure to be used to pass attributes to created threads.
pthread_attr_set<attr>	Sets a certain attribute in the structure.
pthread_attr_get<attr>	Gets a certain attribute in the structure.
pthread_attr_destroy()	Destroys attribute structure.
pthread_self()	Returns the identification number of the calling thread.
pthread_exit()	Terminates the execution of the calling thread.
pthread_join()	Waits until a particular thread exits and returns the exit value of the thread

As threads are operated in a shared memory environment, the communication between processors is straightforward; the shared memory can be used as a communication medium. However, the same method cannot be applied to a distributed memory environment. Therefore, a special library is required to provide communication routines in order for processors to exchange data. The message passing library MPI is explored in the next chapter.

MPI

The Message Passing Interface (MPI) is a library that provides functions to handle communication between processors in a distributed memory environment. However, the use of MPI is not restricted to the distributed memory environment. Most shared memory implementations of the MPI library simply use the shared memory as a communication medium instead of the high-speed network in the distributed memory environment. Nevertheless, the communicated data is packed and handled in a similar way as in the distributed memory environment. Therefore, the justification for the use of MPI in a shared memory can be questioned due to the overhead created by the implementation.

In a distributed memory machine, a process is created for each processor taking part in the calculation. The creation of processes is not done within the code by using MPI. In most cases, the processes are created with a separate program that starts the processes on all the processors specified by the user. All the processes have the same code, which has to be modified so that each processor knows exactly what part of the code to execute. This is usually done based on the process identification number. Since MPI is only a message passing interface, i.e. a communication library, the responsibility of a software developer is to initialize the communication environment, to implement the data exchanges, to synchronize the execution, and to terminate the communication environment.

The MPI Programming Model

Although MPI is not responsible for creating the processes, before MPI can be used, a message passing environment needs to be initialized. The message passing environment contains all processes that were created previously. The initialization of the message passing environment is done with *MPI_Init()* command. At the end of the computation the environment has to be terminated with *MPI_Finalize()* command. Figure 4 illustrates the use of the initialization and termination commands.

```
int argc;
char *argv[];

MPI_Init( &argc, &argv );

MPI_Terminate();
```

Figure 4. Syntax of required MPI commands

As a message passing interface, the MPI offers a large number of commands for point-to-point and broadcast types of communications. The MPI is a very high-level interface, and, therefore, does not require the developer to consider the topology of the interconnection network, packet ordering, or any aspects regarding the transmission and reception of the data. Synchronization is provided by the MPI library with a simple barrier synchronization model. Table 2 lists the basic communication and synchronization routines offered by the MPI library.

With the commands presented in Table 2, all necessary communication patterns can be performed. The barrier synchronization controls the execution of processes. The synchronization will be discussed in detail in the next chapter. Similar to the thread programming model, the MPI provides calls to acquire the identification numbers of the processes and the numbers of the processes taking part in the computation. More about the implementation of MPI and MPI commands can be found in (Message Passing Interface Forum 1994; Pachero 1997; Snir and Dongarra 1998).

SYNCHRONIZATION

Synchronization is needed to guarantee the correct order of execution and memory coherence. The dilemma of synchronization does not exist in a sequential application, since a single copy of the application is run one command at a time. Parallel execution introduces the possibility of executing several copies of the application, and making multiple memory references simultaneously. Concurrent memory references can have a significant effect on the correctness of the output; on the other hand, simultaneous memory references can be allowed as long as the address is different.

Table 2. Basic communication and synchronization functions in the MPI library

MPI_Send()	Sends data to a specific process.
MPI_Recv()	Receives data from a specific process.
MPI_Bcast()	Sends data to all processes.
MPI_Reduce()	Received data from all processes and process data.
MPI_Scatter()	Distributes data between processes.
MPI_Gather()	Centralized data to a single process.
MPI_Barrier()	Synchronized the execution of processes.
MPI_Comm_rank()	Gets the identification number of calling process.
MPI_Comm_size()	Gets the number of processes in the message passing environment.

Based on the description of synchronization, two synchronization models can be identified: the synchronization of execution, and the synchronization of memory references. The distinction between the synchronization models is required since the implementation methods of the models are different. For the synchronization of execution usually a single command is required to halt the execution of a process or thread until a certain condition becomes true. The simplest and most popular implementation of the synchronization of execution is barrier (MPI_Barrier for example). The barrier command in the MPI is implemented so that the execution of a process stops until all the other processes have reached the barrier command. Then the execution of all processes continues simultaneously. To protect a memory address from concurrent access, only one process or thread is allowed to refer to the memory at a time. This is implemented by creating a protected area in the code. A process or thread needs to acquire a lock before entering the protected area, and since only one process or thread can hold the lock and enter the protected area the lock is called a mutual exclusion lock.

In simulators, it is essential that the events are processed in the correct order, and that certain events are fully processed before new events can be handled. The correct order of execution can be guaranteed with mutual exclusion locks. The locks will ensure that no two processes or threads handle the same simulation object or refer to same memory location simultaneously. On the other hand, the handling of new events can be deferred with the use of a barrier, which guarantees that all processes have finished their jobs before new events are processed.

One should realize that since synchronization is not required in a sequential application, synchronization is considered overhead. Overhead is the extra computation introduced due to the simultaneous execution of an application. The simulation expert should consider the use of synchronization models carefully and choose the one that generates a minimal

amount of overhead. It has been shown that incorrect synchronization models and their implementations have a significant effect on the performance of the application.

For example, the GSM network simulator uses both kinds of synchronization models. After the vertical calculation is completed, processors stop at a barrier until all processors have completed their work. Then, the simulation continues with horizontal calculation. Horizontal calculation updates the values of global variables that store the results of the simulation. Since it is possible that multiple processors access the same global variable, and add a new value to it, a synchronization of memory references has to be carried out. The part of the code that updates the values of the global variables is defined as a protected area, which is protected with a mutual exclusion lock. Therefore, only one processor can access the area and update the values of the global variables at a time.

LOAD BALANCING

Since the application is executed on multiple processors, the application needs to be divided into parts, which are then distributed to the processors. The purpose of load balancing is to divide the workload into optimal proportions with respect to the sizes and the capabilities of the processors. There are a great number of factors that affect the implementation of a load balancing algorithm. Firstly, the algorithm requires information about the workload; the workload may be static or dynamic. Secondly, based on the characteristics of the workload, a basic division of work can be done. However, the effects of the environment must also be considered; how are the processors connected together? Do all processors have the same clock rate and an equal amount of memory? The topology of the interconnection network dictates the communication pattern and delay, whereas the differences in processor type and memory affect the amount of work given to a certain processor. In addition, the load balancing algorithm requires information about the current load of each processor. Therefore, the load balancing clearly depends on multiple factors. In most cases, it is not possible to gather all the information mentioned above; nevertheless, close to optimal load balancing algorithm can be implemented taking into consideration the information that is known to have a significant effect in a specified environment.

The importance of load balancing should not be underestimated under any circumstances. It is crucial that a correct load balancing algorithm is implemented, and that the algorithm reflects the special aspects of the environment where the application is run.

For example, in a GSM network simulator the workload consisted of line-of-sight polygons. The differences in the polygon sizes were of several orders of magnitude. Since the execution times of the polygons were proportional to their sizes, the distribution of execution times was large. Therefore, the load balancing algorithm had to distribute the work in such a way that the amount of work per processor was equal, instead of the number of the polygons. Since a heuristic and time-consuming algorithm would have been required to optimize the amount of work per processor, a work pool was created with all the polygons in it. Each processor removed the biggest polygon from the pool and processed the polygon. This was repeated until the pool was empty. Since the implementation of the GSM network simulator was done in a shared memory environment, the issues related to communication were ignored. However, the load balancing algorithm required a synchronization model to guarantee that the same polygon was not removed by two or more processors; so, the access to the pool was protected by a mutual exclusive lock.

PERFORMANCE ISSUES

Since the programming models differ from each other and are optimal for different kinds of environments, the comparison of performance is difficult. Due to the fact that threads cannot be used in a distributed memory environment, a shared memory environment was chosen as a test platform. The communication with MPI used the shared memory as a communication medium as opposed to the high-speed network used in distributed memory environments. Therefore, message passing times in the MPI implementation did not show the total time spent in message passing in distributed memory environments. To compensate for this fact, a comparison between MPI implementations for shared and distributed memory environments was also conducted.

The test programs measured the time spent in communication between two processors. Even though synchronization is the other contributor to overhead in parallel computing, the effect of synchronization is not as significant as communication. To test the performance of the MPI library, the master process created a large amount of random numbers. These random numbers were then sent to the slave process, which sent the numbers back to the master. The communication time was measured as a round-trip time from master to slave and back. The round-trip time gives a better indication of the total time spent on communication. If only the one-way time is measured, the communication time may reflect the time spent in preparation of communication. The size of a packet, that is the number of random numbers, was varied from 1 to 15 million. To acquire reliable results statistically, a great number of repetitions were performed with each packet size.

Since threads are used in a shared memory environment, explicit message passing is not required. However, it is not always possible for two or more threads to share a variable, since both of the threads may want to modify the variable at the same time. The solution for preventing concurrent access is to use synchronization, although, synchronization can be ignored if two copies of the variable exist. The approach of two separate variables was taken in the test program for threads. As explicit sending is not needed, the first thread indicated to the receiving thread by using a signal that a copy of the variable can be made. After the receiving thread has made the copy, it sends a

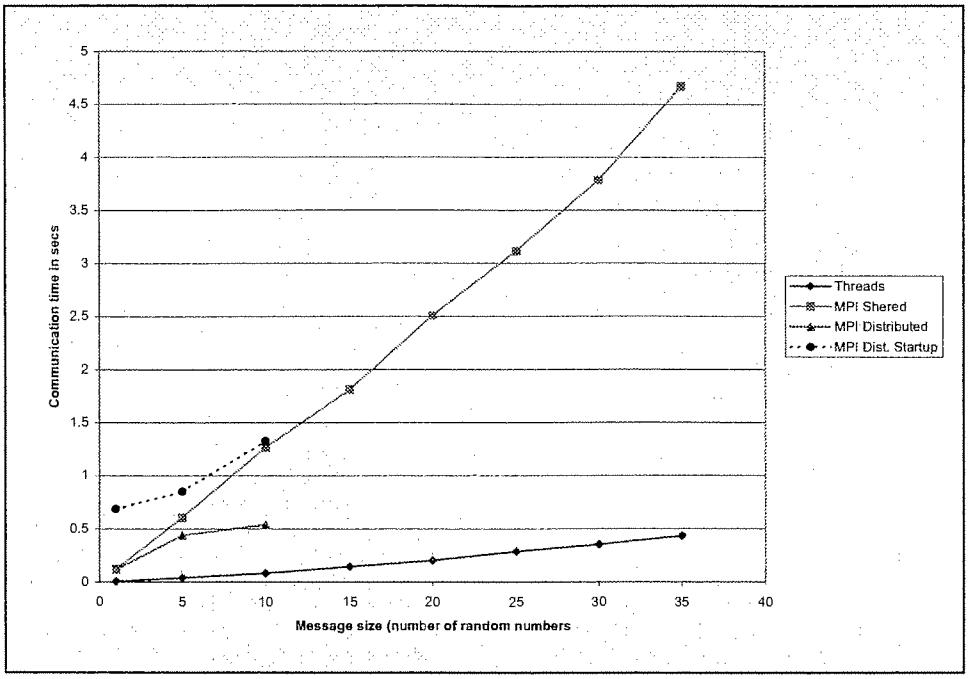


Figure 5. Execution times of the test programs.

signal to indicate that a copy has been made. Finally, the first thread makes another copy of the variable. This test scheme simulates the message passing implemented in the MPI test program. The packet sizes and repetitions were varied similar to the MPI test.

Figure 5 shows the execution times of all the test programs. The distributed memory environment did not have more than 128 megabytes of memory per processor, which limited the number of random numbers that could be used as workload. The number of random numbers was varied from 1 million to 15 million, whereas in the shared memory environment the maximum number of random numbers was 35 million. The comparison between threads and MPI shared shows that threads consume considerably less time in communication. The reason for this phenomenon is that signals require far less computation that the MPI send and receive routines. The MPI routines work similarily depending on the environment. The sent data is packed before sending and unpacked after receiving regardless of being shared memory environment. As for the distributed memory environment, the communication time was shorter than the corresponding time in the shared memory environment for the first three test cases. However, the comparison between the two MPI implementations is not quite acceptable. The MPI implementations were run on machines that had different kinds of processors and interconnection networks. In general, the shared memory implementation should be faster than the implementation for a distributed memory environment, due to the differences in the communication media. The dotted line in Figure 3 shows the time required to send the first packet of random numbers in a distributed environment. As can be seen, the time spent in sending the first packet in the distributed memory environment is greater than the average time in the shared memory environment. However, the average communication time in the distributed memory environment is less than the average in the shared memory environment, since the communication time decreases considerably after the first packet. Prior to sending the first packet the MPI has to initialize the communication in the distributed memory environment; the initialization consists of determining the location of the receiving processor, and determining the route to the receiving processor. The subsequent packets do not require initialization since the packets were sent to the same processor. Therefore, the communication times of the subsequent packets were considerably shorter than the first packet. This led to an average time that was shorter than the corresponding MPI test in the shared memory environment.

Even though the communication time with threads is relatively short compared to any of the other test cases, the use of threads is limited to a certain number of processors. Since threads share the memory and the communication and synchronization are both handled via the memory, the memory bus will eventually become congested. Therefore, the maximum number of processors in a shared memory environment is usually limited to 20. If more than 20 processors are required, a distributed memory environment is a better choice. The biggest distributed memory environment consists of thousands of processors connected to each other with a high-speed network. The full potential of the MPI and communication between processors can be seen when the number of processors is considerably high.

CONCLUSIONS

In this paper, two commonly used programming models for multiprocessor environments were presented. The goal of the paper was to introduce simulation experts to the programming models and their characteristics. The thread programming model for shared memory environments was discussed. The usage of threads is relatively easy; only one command is required to take advantage of threads. However, since threads can be efficiently applied to a shared memory environment, the Message Passing Interface (MPI) was presented as an alternative programming model for distributed memory environments. The MPI provides commands for processes to communicate with each other. Implementation with the MPI is easy requiring a handful of commands. The mandatory commands deal with initialization and termination of the communication environment. Following the discussions on programming models, two important issues of parallel computing were covered: synchronization and load balancing. Synchronization is required to guarantee the correct execution of the code and to prevent simultaneous memory references to same memory locations. The responsibility of a load balancing algorithm is to provide an equal division of work between processes. The algorithm has to consider numerous factors that have an effect on load balancing, and eventually, the performance of the parallel application. During the presentations of synchronization and load balancing, examples were given in general and from the GSM network simulator. The parallel versions of the simulator were implemented with the threads and the MPI to shared and distributed environments. Finally, a study of the message passing between processors was conducted in the shared and distributed memory environments. The study showed that the implementation with threads is faster than any implementation with the MPI. On the other hand, the thread implementation was done in a shared memory environment, which provided a more suitable environment for message passing in a form of the shared memory. However, it should be noted that the distributed memory environment is not a much slower.

To conclude, the utilization of multiple processors is a relatively simple operation if the software developer is aware of a few important issues related to parallel computing. Both the programming models presented in this paper are high-level models. Therefore, the use of the models is easy and straightforward. However, the issues concerning the memory environment, synchronization and load balance should be addressed carefully.

REFERENCES

Butenhof, D.R. 1997. *Programming with POSIX Threads.* Addison-Wesley. ISBN 0-201-63392-2.

Lewis, B.; D. Berg. 1996. *A Guide to Multithreaded Programming. Threads Primer.* SunSoft Press. ISBN 0-13-443698-9.

Message Passing Interface Forum. 1994. "MPI: A Message Passing Interface Standard." *International Journal of Supercomputer Applications* 8 (3/4).

Pachero, P.S. 1997. *Parallel Programming with MPI.* Morgan Kaufmann. ISBN 1558603395.

Snir, M.; J. Dongarra; et al. 1998. *MPI: The Complete Reference: MPI-1 Core.* MIT Press. ISBN 0262691841.

BIOGRAPHY

Pentti Huttunen received his Master of Science degree from Lappeenranta University of Technology in 1999. Currently his is pursuing a Ph.D in parallel and distributed computing while working as a research engineer at Lappeenranta University of Technology. His special interests comprises of thread programming, work balancing algorithms, and cluster computing.

Jouni Ikonen received Master of Science degree from Michigan Technological University in 1994, Master of Science degree in 1995 and Licentiate of Technology in 1999 from Lappeenranta University of Technology. He is currently working as research engineer at Lappeenranta University of Technology. His interests include distributed simulation, wireless networks and data communications.

Jari Porras received Master of Science degree from Michigan Technological University in 1993, Master of Science in 1993, Licentiate in Technology degree in 1996 and Doctor of Technology degree from Lappeenranta University of Technology in 1998. Currently he is working as a professor at the Lappeenranta University of Technology. His main research interests are the use of clustered workstations for distributed computing, parallel algorithms and cellular networks.

Publication 4

Huttunen P, Ikonen J., and Porras J.: Enhancing Load Balancing in a Data-Parallel GSM Network Simulator through Application-Specific Information. In Proceedings of Conference on Applied Parallel Computing, Helsinki, Finland, June 15-18, 2002, pp. 542-554.

Enhancing Load Balancing in a Data-Parallel GSM Network Simulation through Application-Specific Information

Pentti Huttunen, Jouni Ikonen, and Jari Porras

Lappeenranta University of Technology, P.O.Box 20, FIN-53851 Lappeenranta, Finland
{Pentti.Huttunen, Jouni.Ikonen, Jari.Porras}@lut.fi

Abstract. Load balancing plays an important role in achieving good performance in any parallel application. In order to produce a proper load distribution and to minimize overhead, a load balancing algorithm needs to take into account characteristics of the application. Three load balancing algorithms that have been enhanced to optimize the performance of a data-parallel simulator are presented. The enhanced load balancing algorithms are provided with additional information about the workunits, such as an estimate of computational requirements. Results show that the additional information has a significant impact on the load balancing algorithms. The algorithms are capable of producing a more optimal load balance, which leads to the improved performance of an application. A closer study of the overhead created by the processing of the information by the load balancing algorithms indicates that the improvement in the performance significantly outweighs the negative impact of the overhead in most cases.

1 INTRODUCTION

A proper load balancing algorithm is a requirement for good performance results in any parallel application. Since load balancing as a term can be understood in a number of ways, we have made the following interpretations: The term *load balancing* is considered to be a task where workunits are distributed among available processors. It is assumed that the partitioning of the *workload* into *workunits* has been performed prior to applying the algorithms described in this paper. The task of the load balancing algorithms is to either statically or dynamically allocate the workunits among the processors. The distinction between a static and dynamic load balancing algorithm is in timing of the load balancing decisions. A static algorithm performs the distribution prior to a computation phase, whereas a dynamic algorithm distributes the workunits during the computation phase based on the current system situation.

The basic concept of load balancing in parallel and distributed systems is well known. A number of articles have been published that discuss different aspects of load balancing [1][2][3][4]. Numerous load balancing algorithms have been proposed that are intended either for solving a specific problem in a specific system or for the theoretical analysis of load balancing in a variety of systems [5][6]. In the latter

category there are some very interesting proposals that are applicable to data-parallel applications. The majority of the algorithms assume that the computational requirements of a workunit being distributed do not have a significant contribution to load balancing decisions. The algorithms give more emphasis to current loads of the processors participating in the computation. Thus, the load balancing decision is often made by looking at the current load situation, and selecting either the least loaded or the most powerful processor [7][8].

Another approach for load balancing is to consider the computational requirements of workunits. This approach allows the workunits to be distributed among the processors in multiple ways depending on the system. For example, the workunits can be distributed so that each processor receives an equal amount of work to process or the amounts of work for processors are proportionally distributed among the processors according to their capabilities. As such, a load balancing algorithm distributing the workunits among the processors based on the computational requirements of the workunits does not consider the current loads of the processors. However, this problem can be solved by implementing a global client-server type load balancing algorithm. The third algorithm discussed in this paper shows the implementation of such an algorithm.

In this paper three generic load balancing algorithms are enhanced by providing them with additional information about the application and, especially, the workunits. The information is extracted at run-time from the workunits to allow the algorithm to adapt to changes in workloads. In the rest of the paper, the method of load balancing, utilizing the additional information, is referred to as application-specific load balancing. Naturally, the purpose of the application-specific information is to refine the load balancing decisions by allowing the algorithm a better view of the workload. Furthermore, the load balancing algorithms introduced are the most suitable for data-parallel applications, where all processors execute the same code.

A GSM network simulator was used as a case study for the algorithms. The simulator is used by mobile network providers to design their networks for urban microcell environments. The providers are able to determine optimal locations for their base stations based upon the results given by the simulator. Thus, the simulator computes a coverage area of a base station over a map consisting of buildings and streets.

The structure of the paper is as follows: Section 2 talks about data-parallel computing by introducing the concept and its benefits. Section 3 discusses the GSM network simulator used as a case study for the three load balancing algorithms. The three load balancing algorithms with their application-specific versions are introduced and the results achieved are shown in Section 4. Section 5 discusses the results and implementation issues of the application-specific load balancing algorithms. Finally, the conclusions are drawn in Section 6.

2 DATA-PARALLEL COMPUTING

Data-parallel computing can be defined in multiple ways [4]. The lowest level defines a data-parallel application to be one in which the instructions and data are distributed by a control unit. This control unit dictates what is processed on any given processor at any given time. The execution usually takes place in a synchronized fashion. Another, and more relaxed, definition of a data-parallel application is an application where each process executes the same code with different data. In this case the synchronized execution of processors is not mandated, so the processors can freely process data assigned to them. In order for the processors to process data without synchronization there must not be any dependencies between the data. If data dependencies exist, the processors require synchronization of their execution or/and communication with each other. The latter definition of a data-parallel application is applicable to the simulator discussed in this paper.

The environment in which the data-parallel application is run has an impact on the design and implementation of the application and load balancing algorithm. In a shared memory environment the processors share the memory allocated for the application (process). Therefore, the synchronization and communication is carried out through the shared memory. In general, the utilization of the shared memory in synchronization and communication is very fast. The congestion in the network that connects the processors to the shared memory can cause a problwm, which can be avoided by limiting the number of processors in the system. In a distributed memory environment the situation is quite different. Even without any data dependencies there is a need for communication. This communication occurs when the data is distributed among the processors as well as when the results are gathered from the processors for the processing of the final results. In addition to the previously mentioned mandatory communication, the synchronization also requires communication. As previously examined, the communication is a significant contributor in the distributed memory environments to the overall performance of a data-parallel application. The advantage of the distributed memory environments lies in the ability of the systems to facilitate a substantially bigger number of processors than shared memory systems.

Even though sometimes the input data has no dependencies, the output (result) data creates dependencies between processors. It is possible that the results computed from the data are stored into a shared memory. Multiple pieces of data may cause the same element of results to be updated. In the worst case, two or more processors update the result element simultaneously. Concurrent access cannot be allowed in order to guarantee the correctness of the results. Therefore, a synchronization method has to be put into place to guard the results from concurrent access.

Data dependencies are not the exclusive cause of communication. A load balancing algorithm responsible for distributing the data among the processors resorts to a communication medium to transfer the necessary data to processors. The algorithm has to send pieces of data to each processor according to the load balancing algorithm. Therefore, the cost of the communication has to be considered a factor in designing and implementing a proper load balancing algorithm.

A load balancing algorithm can have multiple tasks. It might be up to the load balancing algorithm to divide the workload into suitable workunits, and only after this is complete to perform the distribution based on the actual algorithm. We identify the division of the workload to be partitioning. Obviously, if the data can be divided so that no dependencies exist, the performance of the application can be greatly increased due to the lack of communication and synchronization. However, details about partitioning are out of the scope of this paper. The algorithms presented in this paper assume that the workunits have been determined prior to the execution of the load balancing algorithm. Next, the workunits are distributed among the processors, i.e. the actual load balancing is performed. There are a number of load balancing methods to allocate the workunit. Three load balancing algorithms for data-parallel applications are discussed in this paper. A possible problem caused by the application is the sizes of the workunits identified by a partitioning algorithm. It is quite likely that in complex data-parallel applications the workunits are not equal in size. Furthermore, if the workunit size is directly proportional to the computational requirement of the workunit, the load balancing algorithm has to be able to compensate its actions based on the workunit sizes. On the other hand, in a heterogeneous computing environment the processing capabilities of the processors are different. Both of these situations complicate the work of a load balancing algorithm. For optimal performance, the load balancing algorithm has to consider the workunits in greater detail than by merely performing the distribution based on the number of workunits. We have enhanced three basic load balancing algorithms to take into account application-specific information to further refine the load balancing operations. The application-specific information has a dramatic impact on the performance results of the algorithms, as will be been seen in the following sections. However, prior to describing the load balancing algorithms with their application-specific counterparts, the data-parallel application, the GSM network simulator, used in the experiments is introduced.

3 THE GSM NETWORK SIMULATOR

The GSM network simulator is used in network planning to find optimal locations for base stations in microcells [9]. The simulator considers signal propagation in a 3-dimensional space by utilizing ray tracing methods. The simulation computes the coverage of a base station over a map of buildings and streets. The map is divided into receiving points that hold the coverage information. A field strength value of a receiving point is updated whenever a signal arrives to that particular receiving point. The simulation model (ray tracing) is enhanced by introducing a concept of line-of-sight (LOS) polygons[9][10]. A LOS polygon is generated at each corner of a building indicating the diffraction area of a ray. When a ray arrives at a corner, it diffracts over the area defined by the LOS polygon. Therefore, the actual simulation phase comprises of the processing of the LOS polygons. For each receiving point that falls inside a LOS polygon the following operations are performed. The strength of the signal at the receiving point is calculated as a function of the initial signal strength

at the corner and the distance from the corner. The new signal strength value is updated to the structure that holds the field strength information of all receiving points.

3.1 Parallelizing the GSM Network Simulator

The LOS polygons form the workload that is distributed among the processors. Since the size of a LOS polygon depends on the layout of the buildings, the polygons are of different size. Fortunately, it has been determined through testing that the size of a LOS polygon is directly proportional to the computational requirements of the polygon. Therefore, the workunit size and computational requirement are interchangeable when referring to the workunits (LOS polygons).

In a shared memory environment where the processors (threads) share a global memory, the signal strength information for all receiving points are kept in memory locations accessible by all processors. The LOS polygons are independent workunits, but it is possible that two or more processors update the same receiving point simultaneously. This is due to the fact that the LOS polygons can overlap each other. In order to guarantee that no more than one processor updates any given receiving point at a time, a synchronization method needs to be implemented. The synchronization method prevents more than one thread from accessing the same receiving point simultaneously. Other processors are forced to wait for their turn, if concurrent access is attempted. The necessary synchronization introduces additional computation (overhead), but its contribution to the overall execution time of the data-parallel simulator is minimal.

In a distributed memory environment the situation is somewhat different. There is no need for synchronization, since each processor has a copy of all the receiving points. The receiving points are updated locally, and gathered at the end of the simulation to a processor that produces the final results. On the other hand, in a distributed memory environment the load balancing requires communication in addition to the gathering of the final results. The load balancing algorithm communicates with processors in order to distribute the LOS polygons. The communication is the largest contributor to overhead in the distributed memory environment.

4 LOAD BALANCING ALGORITHMS

Three generic load balancing algorithms are introduced that distribute the LOS polygons among the processors. These algorithms have limited knowledge of the workunits and are only provided with the number of workunits (LOS polygons). Next, the algorithms are enhanced with the application-specific information. In the case of the GSM network simulation, the application-specific information is the computational requirements of the LOS polygons. In general, if the algorithms are provided with information about the computational requirements of workunits, algorithms can be utilized more efficiently. Performance results are given for both

versions of the algorithms in order to quantify the improvements achieved with the application-specific load balancing algorithms.

4.1 Load Balancing Algorithm 1

The first algorithm distributes the workunits so that the number of workunits is equal among the processors. If the number of workunits is not divisible by the number of processors, the last *r* processors have one more workunit to process. The algorithm is not computationally very complex. Furthermore, the algorithm is capable of producing a good load balance, if the workunits are all equal in size and the processors are homogeneous. If the workunits are not the same size, the algorithm cannot produce an optimal load balance without receiving additional information about the workunits.

The application-specific version of the load balancing algorithm 1 utilizes information about the sizes of the workunits, i.e. the computational requirements. The algorithm requires an array where the sizes of the workunits are stored. The array is sorted in descending order based on size. The workunits are assigned to processors so that each processor receives an equal amount of work, not necessarily an equal number of workunits. It should be noted, that it is highly unlikely that all the processors will have exactly the same amount of work to do resulting from the differences in workunit sizes. However, the algorithm optimizes the distribution so that each processor receives an amount of work as close to the average as possible.

Even though the algorithm allocates the last processor the remaining workunits regardless of the amount of work they involve, the algorithm produces equal work distributions. This is a result of the optimization that takes place when the previous processors are assigned the workunits. The algorithm allocates workunits so that the amount of work each processor receives is either less or more than the average, whichever is the closest. The optimization improves the load balance and reduces the possibility that the last processor will receive considerably more work than the other processors. The application-specific load balancing algorithm introduces overhead: computing the total amount of work and sorting the workunits. However, more optimal load balancing and the increased utilization of the processors compensate for the overhead.

Fig. 1 shows a comparison of the execution times of the GSM network simulator run with the two versions of load balancing algorithm 1 on a network of workstations consisting of 7 Pentium III workstations running the Linux operating system. The figure illustrates the superiority of the application-specific algorithm over the generic algorithm. The application-specific algorithm is able to produce more equal load balances among the processors due to the fact that it has knowledge about the computational requirements of the workunits. When the number of processors is increased, the benefits of the application-specific load balancing algorithms become especially evident.

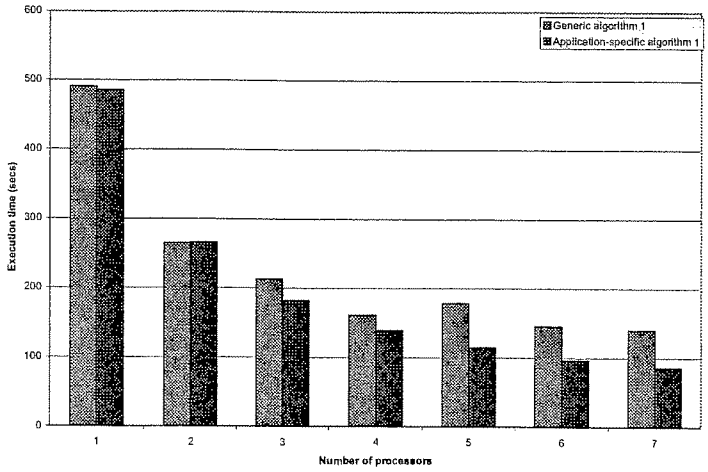


Fig. 1. Execution times of the simulator with generic and application-specific algorithms 1

Even though the application-specific load balancing algorithm performs substantially better than the corresponding generic algorithm, there is still room for improvement. Possible reasons for load imbalance among the processors still existing are: (1) the workunits have vastly different computational requirements. This leads to situations where certain processors have to process a large number of "small" workunits, whereas other processes are left with a relatively small number of "large" workunits. It is more time-consuming to process a large number of "small" workunits than a small number of "large" workunits, due to the overhead created by the retrieval and initialization of the workunit. (2) The workunits cannot be distributed equally so that each processor has exactly the same amount of work to do. This unavoidably leads to a situation where the execution times of individual processors vary.

4.2 Load Balancing Algorithm 2

The second algorithm is based on the assumption that the workunits are uniformly and independently distributed over the array where they are stored. The algorithm picks a workunit at a fixed interval starting from the beginning of the array. The first workunit for a processor is determined by its rank. For example, the first processor (having a rank of 0) picks the first one, and so on. After that each processor jumps over p workunits to reach the next workunit to process, where p is the number of processors participating in the computation. The preceding step is repeated until each processor reaches the end of the workunit array. Due to its simple implementation, the algorithm does introduce very little overhead.

The application-specific load balancing algorithm accounts for the differences in the sizes of the workunits by sorting the workunits prior to the distribution. The sorting operation guarantees that the largest workunits are processed first by all the processors. This, in turn, improves the processor utilization at the end of the

computation. The processors are more likely to finish at the same time, since the last workunits are relatively small.

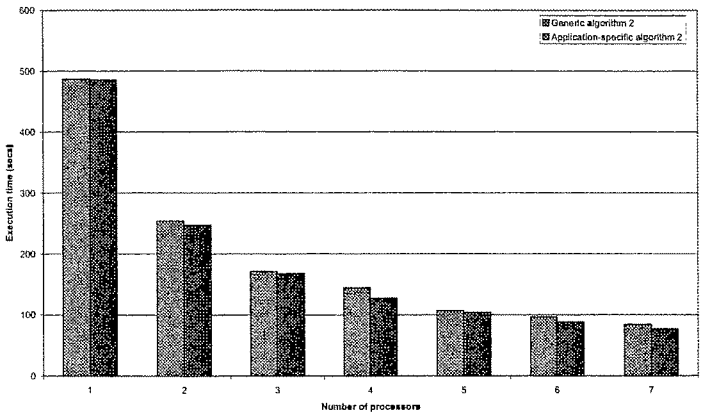


Fig. 2. Execution times of the simulator with generic and application-specific load balancing algorithms 2

Fig. 2 illustrates the execution times achieved with the two versions of the load balancing algorithm 2. Again, the application-specific algorithm performs better. In addition to having more information about the workload, the application-specific algorithm is more efficient, since the assumption that the workunits are uniformly and independently distributed is not quite true in the case of the GSM network simulator. The workunits are not distributed uniformly, whereas they seem to be clustered in certain parts of the array. Regardless of the incorrect assumption, the algorithm performs better than the generic load balancing algorithm 1; the worst-case scenario of the generic algorithm 2 is approximately 20% better than the best-case scenario of the generic algorithm 1. The increased performance is derived from the better (more equal) utilization of available processors. The application-specific load balancing algorithm 2 produces approximately 35% better workload distribution than the application-specific algorithm 1.

4.3 Load Balancing Algorithm 3

The third load balancing algorithm creates a pool of workunits, and dynamically allocates the workunits to processors as per their requests. A single processor is dedicated to act as a server that receives the requests and sends the workunits as replies. The algorithm sends a single workunit at a time to any requesting processor. The performance of the algorithm can be improved further by sending an index of the workunit, instead of the workunit itself. This approach requires that all processors have all the workunits in their local memories, and that the workunits can be uniquely addressed by an index. In general, the purpose of the algorithm is to optimize the utilization of the processors by allocating only one workunit at a time to each individual processor when it asks for more work. The generic algorithm instructs the

processors to process workunits in the order in which they are stored in the workunit array. The application-specific version of Algorithm 3 sorts the workunits based on their sizes and distributes the workunits in descending order to the processors as per their requests.

A comparison of the generic and application-specific algorithms is depicted in Fig. 3. The results are a very interesting in respect to the application-specific algorithm. Based on the execution times of the simulator the generic algorithm is faster when the number of processors is larger than 5. However, workload distribution is considerably better in the application-specific algorithm when the number of processors is larger than 5 (figure not shown due to the limited availability of space). The reason for this somewhat puzzling situation is the fact that the communication between the server processor and the client processors creates a bottleneck. In part, this is true for the generic algorithm as well. However, the impact of the communication bottleneck is emphasized in the application-specific algorithm, since there are a number of concurrent requests sent to the server processor. The use of the application-specific algorithm introduces simultaneous requests due to the order in which the workunits are processed. As mentioned, the server processor assigns the workunits in descending order based on their computational capabilities. Therefore, at any given time most of the client processors are processing workunits that are very close to each other in terms of size. This leads to a situation where a number of processors have issued requests to the server processor simultaneously. The sending of the requests does not have to take place exactly at the same time because it takes some time for the server processor to handle the request and send back the reply. Since the server processor is capable of processing a single request at a time, the other possible requests are queued and they have to wait their turn in order to be processed.

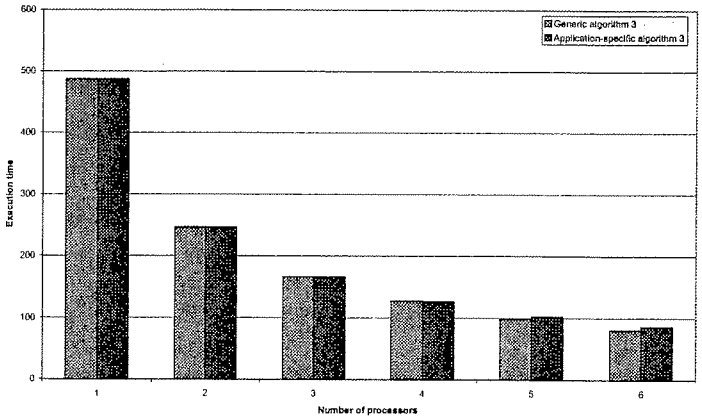


Fig. 3. Execution times of the simulator with generic and application-specific load balancing algorithms 3

Due to the above-mentioned reasons, the generic algorithm provided the best performance results out of the two load balancing algorithms. Overall, algorithm 3 (generic version) was the best performing algorithm of all the three different

algorithms studied. The algorithm 3 performed approximately 22% and 21% better than the best cases of application-specific algorithms 1 and 2, respectively. It still should be emphasized that application-specific algorithm 3 was able to produce a better load balance, but suffered from the communication bottleneck and the centralized scheduling mechanism.

4.4 Comparison of the algorithms

Each of the algorithms and their corresponding application-specific implementations make assumptions about the workload. The first algorithm assumes that the workunits are similar in their computational requirements. The algorithm 1 is, indeed, a good solution for load balancing of workunits that take an equal amount of time to process. The algorithm is able to produce a good load balance with minimum amount of overhead. In the GSM network simulator, the problem with the algorithm 1 proved to be the fact that the computational requirements of the workunits were vastly different. Not even the application-specific algorithm 1 was able to compensate for the differences efficiently. The algorithm 2 expects that the workunits are equally distributed with respect to their computational requirements over the array in which the workunits are stored. This approach tries to optimize the load balance based on statistical assumptions that the workunits are uniformly and independently distributed. In an ideal situation, the algorithm 2 can produce an equal load balance among the processors while incurring very little overhead. In the case of the GSM network simulator, it was noticed that the workunits were not independently distributed. This was due to the fact that the workunits (LOS polygons) were generated based on a map. The map created dependencies between the workunits causing clusters of computationally expensive workunits to form. The algorithm 3 is a general-purpose algorithm that is suitable of all kinds of workloads. However, the generic nature of the algorithm comes with a price. The client-server infrastructure is more complicated to implement, and introduces more overhead than the other two algorithms. On the other hand, the algorithm proved to be the best from the performance and load balance point of views.

The performance results of the generic algorithms are shown in Fig. 4. The figure illustrates total execution times of the simulator, whereas the previous figures have merely shown execution times of the parallel computation phase of the simulation. The performance results of the whole simulation are impacted by a sequential component of the simulator; the significance of the sequential component becomes more obvious when the parallel computing phase gets shorter. Algorithm 3 has the best performance of the three algorithms. At best, algorithm 3 provided performance improvements of 81% and 19% compared to algorithm 1 and algorithm 2, respectively. The corresponding performance improvements for the parallel computation phase were 87% and 20%. The slight decrease in performance in the total execution time is caused by the sequential component of the simulator.

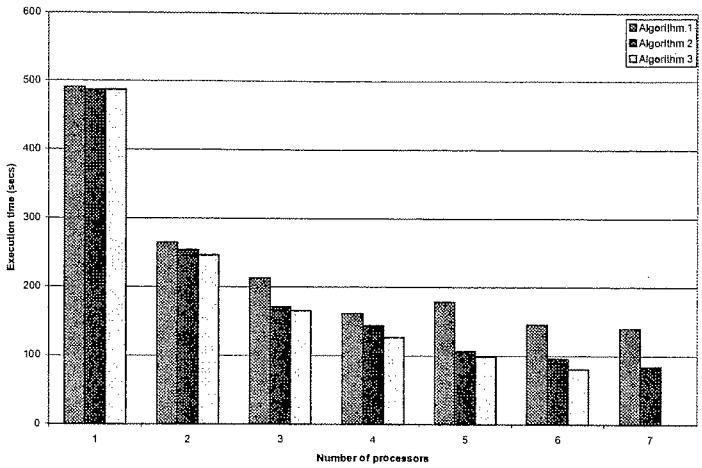


Fig. 4. Execution times of the GSM network simulator with generic load balancing algorithms. It should be noted that no result is given for algorithm 3 in case 7 processors are used. The algorithm 3 requires a master processor that does not process workunits. Therefore, the maximum number of processors available for parallel processing is 6

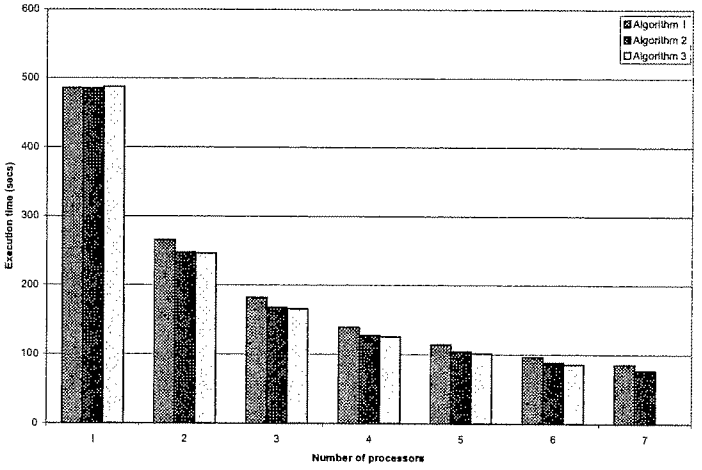


Fig. 5. Execution times of the GSM network simulator with application-specific load balancing algorithms

Fig. 5 depicts a comparison of the three application-specific algorithms in the GSM network simulator. The figure shows that the algorithm 3 performs the best. However, the sequential component of the simulator has also an impact on the overall performance of the simulator. The performance of the load balancing algorithm 3 is 12% and 3% better than application-specific algorithms 1 and 2, respectively. The corresponding performance improvements for the parallel computing phase were 13% and 3%.

5 DISCUSSION

In general, the application-specific load balancing algorithms performed better than the generic algorithms from a load balance point of view. In all the algorithms some overhead was introduced by the additional computation required to process the application-specific information, by the communication, and in the case of the Algorithm 3, by the contention in the server processor. However, the computation did not affect the overall performance of the application negatively, since the gain from a better load balance was significant. The communication had an impact on the overall performance of the third application-specific algorithm due to the communication pattern of the processors. The algorithm is a good example of how the communication can form a bottleneck even though the algorithm itself is capable of producing optimal results. A solution to the dilemma would be to create multiple server processors. However, this would require modifications to the algorithm to distribute the server's functionality, and to enable the client processors to determine to which server processor they should send requests. In addition, the server processors are not used for computation. If the number of available processors is limited, assigning processors to server processors would waste computing power. For example, the author's network of workstations consists of 7 workstations. Assigning two processors as servers would reduce the number of processors available for the actual computing to five. Thus, close to 30% of the computing resource would not be performing the actual computation.

6 CONCLUSIONS

Three load balancing algorithms were studied. The generic versions of the algorithms were compared to the application-specific versions in a data-parallel GSM network simulator on a cluster of workstations. The application-specific versions of the algorithms utilized additional information about the workload to further optimize the load balance. In order to take advantage of the application-specific information the algorithms had to do more processing than the generic algorithms: the algorithms had to compute the sizes of the workunits, the average number of work per processor, and sort the workunits in descending order based on their computational requirements. Despite the number of additional operations performed by the application-specific algorithms, the results indicate that the algorithms are able to produce substantially more optimal load balances.

The application-specific algorithms reduced the distribution of the execution times significantly compared to the generic algorithms. In all three algorithms, the application-specific algorithm produces better load balances than the corresponding generic algorithms. However, the third application-specific algorithm performed worse than the corresponding generic algorithm. The problem was caused by the simultaneous communication to and from the server processor. As such this is not a problem with load balancing, since the most optimal results were achieved with the third application-specific algorithm. However, communication was a mandatory part

of the third load balancing algorithm, and could be ignored when overall performance results were compared. After all, one of the goals of parallel computing is to decrease the execution time, which does not necessarily require an optimal load balance among the processors.

In conclusion, the study proved that in order to optimize the load balance and the utilization of the processors, parallel applications require additional information about the workload. Unfortunately, it is not always possible to retrieve such information. However, in cases where the information is available it should be used to improve the capabilities of the load balancing algorithms to produce more optimal workload distributions.

References

1. Diekmann, R., Preis, R.: Load Balancing Strategies for Distributed Memory Machines. Parallel and Distributed Processing for Computational Mechanics: Systems and Tools (1999) 124-157.
2. Deng, X., Liu, H.-N., Long, J.: Competitive Analysis of Network Load Balancing. Journal of Parallel and Distributed Computing **40** (1997) 162-172.
3. Hui, C.-C., Chanson, S.T.: Theoretical Analysis of the Heterogeneous Dynamic Load-Balancing Problem Using a Hydrodynamic Approach. Journal of Parallel and Distributed Computing **43** (1997) 139-146.
4. Norman, M.G., Thanisch, P.: Models of Machines and Computation for Mapping in Multicomputers. ACM Computing Surveys **25** (1993) 263-302.
5. de Doncker, E., Gupta, A., Guo, M.: Adaptive Multivariate Integration using MPI. Proceedings of International Conference on Supercomptuing (1997).
6. Zaki, M.J., Li, W., Parthasarathy, S.: Customized Dynamic Load Balancing for a Network of Workstations. Journal of Parallel of Distributed Computing **43** (1997) 156-162.
7. Ferrari, D., Zhou, S.: A Load Index for Dynamic Load Balancing, Proceedings of XXX, (1986) 684-690.
8. Meyer, T.E., Davis, J.A., Davidson, J.L.: Analysis of Load Average and its Relationship to Program Run Time on Networks of Workstations. Journal of Parallel and Distributed Computing **44** (1997) 141-146.
9. Heiska, K., Kangas, A.: Microcell Propagation Model for Network Planning. Proceedings of IEEE Symposium on Personal, Indoor, and Mobile Radio Communication (1996) 148-152.
10. Sipila, K., Heiska, K.: Can ray tracing be used as a fading generator in simulating micro cellular mobile radio systems? Proceedings of Conference on Wireless Communication (1996).

Publication 5

Huttunen P., Ikonen J., and Porras J.: MPIT – Communication/Computation Paradigm for Networks of SMP workstations. In Proceedings of Conference on Applied Parallel Computing, Helsinki, Finland, June 15-18, 2002, pp. 160-171.

MPIT – Communication/Computation Paradigm for Networks of SMP workstations

Pentti Huttunen[1], Jouni Ikonen[1], and Jari Porras[1]

[1]Lappeenranta University of Technology, P.O. Box 20, FIN-53851 Lappeenranta, Finland
{Pentti.Huttunen, Jouni.Ikonen, Jari.Porras}@lut.fi

Abstract. A need for a more efficient programming paradigm has been prompted by the introduction of networks of symmetric multiprocessor (SMP) workstations. A new programming paradigm for networks of SMP workstations is presented in this paper. The paradigm called MPIT integrates Message Passing Interface (MPI) and POSIX threads. The MPIT paradigm utilizes MPI for communication among the workstations, and uses threads to process the data within a workstation. The communication among the workstations is handled by a dedicated communication thread that runs on each workstation. The communication among the threads is handled through the shared memory. A number of theoretical and practical benefits of the MPIT paradigm are identified, such as communication/computation overlap, increased resource utilization and performance.

1 Introduction

Networks of SMP workstations (referred to as a network of workstations in the remainder of the paper) have received much attention during the past few years. Their benefits over proprietary supercomputers have driven up the number of systems deployed for a number of reasons. First of all, the cost of a network of SMP workstations is far less than that of a proprietary supercomputer with an equal number of processors. Although the performances of high-end supercomputers are better than a network of workstations, the difference is not substantial. The recently released list of Top 500 supercomputers has 48 cluster systems [1]. The best network of workstations is in the 30th spot on the list outperforming several proprietary systems, such as Cray T3E and IBM SP Power3 systems. Second, a network of workstations is more versatile and inexpensive to expand by adding new workstations compared to adding new processor boards on supercomputers. Third, a network of workstation does not have to be homogeneous system, where all workstations are identical, whereas homogeneity is usually a requirement in proprietary supercomputers. The heterogeneity allows the cluster to have a set of dedicated nodes for special purposes, such as I/O, floating point arithmetic, and graphics. However, the heterogeneous system does bring up new problems, especially with scheduling and load balancing.

The programming paradigm for networks of workstations is message passing. There are numerous message passing libraries of which MPI [2][3] and PVM [4] are,

perhaps, the most popular. Both of these message passing libraries operate in a similar way with respect to multiprocessor workstations; the libraries create multiple processes on a workstation, and the communication either goes through the network (the loopback interface), as if the processes reside in two different workstations, or through the shared memory regions [5]. The use of processes compared to threads is more straightforward, but also computationally more expensive; it is more time-consuming to do a context switch between two processes than between two threads. Furthermore, threads share the memory allocated for the process in which they are created. If threads need to communicate, the communication utilizes the shared memory. In practice, the most simple implementation of a communication method in a shared memory is to use global variables. The global variables are accessible by all threads, which requires a synchronization method to be implemented for controlling the access to the global variables.

The purpose of this paper is to introduce a programming paradigm called MPIT (MPI-Threads) that combines MPI and POSIX threads. One of the main goals of MPIT is to provide a simple, and yet powerful, programming tool for networks of SMP workstations. MPIT provides an interface similar to MPI with the exception that the prefix of the function names is MPIT instead of MPI. The implementation of MPIT creates two types of threads. The first type is a worker thread. The worker thread executes the actual code, and issues send and receive commands to exchange data among the workstations. The second type is a communication thread. Each workstation (process) has a single communication thread that handles all communication operations among the workstations. The communication thread receives messages from the network (i.e. from another workstations), and sends and receives commands from the worker threads. The goal is to allow the worker threads to continue their execution immediately after issuing a communication request to the communication thread. The use of worker threads and a communication thread makes it possible for computation and communication to overlap. Furthermore, the worker threads communicate with each other through the memory allocated for the process in which they are running. All thread creation and maintenance operations are embedded and automatically performed by MPIT.

This paper is organized as follows. First, related work done in the field of parallel computing in networks of SMP workstations, and programming tools for such environments are discussed. Second, MPIT is introduced with detailed descriptions of its architecture, capabilities, and usage. Third, theoretical and practical results of MPIT are explored. Finally, the last section concludes the paper.

2 Related work

Networks of workstations as a platform for parallel computation have been studied for a number of years, and there are reports about studies that closely resemble the one presented in this paper. However, none of the works mentioned in this section about previous studies offer quite the same functionality as MPIT does. Main differences in

MPIT are its utilization of MPI as the transport mechanism among workstations, and its capability to perform load balancing automatically while continuing to provide a very simple interface for the programmer.

A multi-threaded MPI implementation is discussed in [6]. The article outlines the work required to make an implementation of MPI, *mpich*, thread-safe and multithreaded thereby improving its performance. The focus of the paper is on how to change the implementation of MPI to support multi-threading rather than how to provide the user with a multi-threaded programming paradigm on top of MPI.

Very interesting research about the integration of MPI and threads can be found in [7]. The authors present a system called Chant that allows for point-to-point communication between threads in different workstations and the use of remote services requests. Chant allows individual threads to communicate with each other, assuming that the sending and receiving threads are aware of the fact that a communication operation is about to occur. This approach differs from MPIT in the way in which the communication is performed. In MPIT, the communication is handled by a communication thread that removes the need for synchronous communication.

In [8] the authors present a programming paradigm for a cluster of SMP workstations, which consists of Pentium Pro processors. The programming paradigm uses active messages for communication [9], and a specially designed shared memory implementation to handle intra-node communication. The utilization of Active Messages makes the implementation attractive from the performance point-of-view. However, the intra-node communication model is quite complex to learn and understand. In addition, the programming paradigm does not offer any load balancing functionalities.

In [10] a programming model called SIMPLE for clusters of SMP workstations is discussed. The model is intended for parallel computing requiring low communication latencies. The high communication performance is achieved through a communication library developed during the course of the research. The methodology allows for data-parallel programming utilizing threads within a workstation, and the developed communication library to interact with other workstations. However, the model does not facilitate heterogeneous computing environments or load balancing, and does not have as simple interface as MPIT.

3 MPIT

The goal of MPIT is to provide a programming paradigm that allows the overlap of communication and computation without major programmer intervention. MPIT combines the MPI message passing interface and POSIX threads to facilitate the communication-computation overlap. Furthermore, MPIT eliminates the need for explicit communication among processors within a workstation due to the use of

threads instead of processes. The interface to MPIT is very similar to the one used in the MPI library. MPIT has most of the point-to-point communication routines currently found in MPI. Even the parameters in the function calls are practically identical. On the other hand, MPIT introduces a set of new commands to control the execution of threads within a workstation. The following subsections discuss the architecture of MPIT, and describe its usage in detail.

3.1 Architecture

The architecture of the MPIT environment in a workstation is shown in Fig. 1. The figure illustrates the architecture inside a process created during the initialization of the MPI environment. The worker threads are threads created by MPIT to perform the actual computation. The communication thread is a special thread also created by MPIT to handle communication among the workstations. The MPI receive and send buffers are internal buffers utilized by the MPI message passing library. These buffers are shown in the picture to illustrate where the communication thread receives and sends data. The Data-In and Data-Out queue are used by the threads to send and receive messages. These buffers are maintained by MPIT.

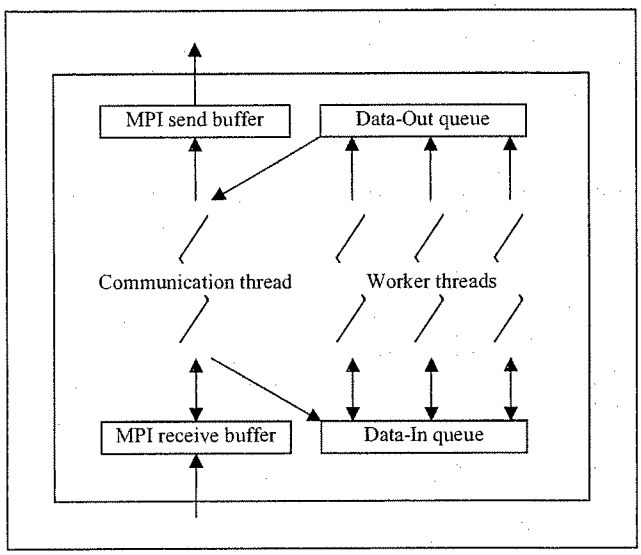


Fig. 1. The architecture of MPIT. This illustrates the components of which the MPIT paradigm is composed. It should be noted that the MPI send and receive buffers are not part of MPIT; the buffers belong to the MPI implementation

When a thread sends data, it calls the appropriate MPIT send function. The MPIT call creates a message and queues it to the Data-Out queue. Once the message has been queued the control is returned to the calling function. From the worker thread's point-of-view the communication has been completed, although the message has not necessarily been sent yet. If a worker thread wants to receive data, it issues a proper MPIT receive function call. The function call causes the data in the Data-In queue to

be searched. If no matching data is found, the execution of a thread is either halted or an error message is returned. It should be noted that the worker threads do not have direct access to the network. The communication always goes through the data queues.

The purpose of the communication thread is twofold; it is responsible for handling data transfers in and out of the workstation, and performing workload scheduling. The communication thread constantly monitors whether the user has issued MPIT send commands or if there is data to be received from the network. If there is data in the Data-Out queue, the communication thread removes the data from the queue and initiates a send procedure. The communication thread sending the data does not have to know that there is a matching receive operation waiting for the data. The thread sends the data assuming that the communication thread in the receiving process either receives it immediately or at a later time. When the communication thread notices that there is data pending in the MPI receive buffer, the thread initiates a receive command to retrieve the data. The communication thread is able to receive any kind of data from any workstation, even from itself. Even though the architecture facilitates communication within a process through the buffers, it is more prudent to use the shared memory for thread communication within a workstation.

Since MPI communication commands require detailed information about the sent and received data, such as number of data elements, data type, sender, and tag, the communication takes place in two phases. When the communication thread removes data from the Data-Out queue, two messages are generated and sent to the network. The first message contains information about the actual message: the type of the actual message as well as the size of the message. With the help of the additional message the other end of the communication is able to receive the message properly. The information message is immediately followed by the actual (second) message. The receiving side's communication thread monitors the information messages. Once the thread has received one of the information messages, it initiates a receive operation to retrieve the actual data. When the message has been received, the thread enqueues the message to the Data-In queue. If there are worker threads waiting for new data to arrive, the threads are notified about the new data.

Another benefit of MPIT in addition to the communication-computation overlap is its ability to handle the dynamic scheduling (load balancing) of workunits in data-parallel applications. The automatic load balancing is based on a client-server model and implemented with a handful of function calls. The programmer determines a master workstation that acts as a server for the other workstations. Threads on all the workstations, other than the master, send requests to the master when they are out of work. The request by a thread for more work is handled similar to any other send operation on the sender's side. However, the data is sent to the master workstation, which was defined during the initialization of the MPIT environment. The communication on the master workstation generates a reply to the work request, and sends it back to the requesting thread. Since the communication thread performs the load balancing operations, no worker threads on the master workstation are affected. The load balancing can be performed in two different ways; the master workstation

can either send a workunit to the requesting thread or an index to a workunit. The latter case assumes that the requesting thread has knowledge of all the workunits. In order for the master workstation to perform load balancing actions, information about the workunits and, possibly, their weights must be given to the MPIT environment. This is done before or after the MPI environment is initialized with a set of MPIT-specific load balancing function calls.

3.2 Setting up the MPIT environment

In order to perform any MPIT operations, the environment must be initialized. It should be noted that the MPI environment must be initialized prior to initialization of MPIT. The following steps are performed during the MPIT initialization:

- A file containing workstation information is read. The file has information about the workstation participating in the computation. For each workstation three attributes must be specified: the name of the workstation, the number of threads to be created, and the relative power of the workstation. The relative power of the workstation is an optional attribute. Its main purpose is to provide information to MPIT about the performance of the workstation. In cases where the cluster is heterogeneous, and MPIT is instructed to perform load balancing, the relative power attribute comes into play. MPIT is capable of distributing the workload among the workstations based on their relative powers.
- Data queues are initialized. The data queues hold the data that is to be transferred and the data that has been received from other workstations.
- Worker threads are created. Each thread begins to execute the function defined by the user in the parameters of the MPIT initialization function. The possible parameters for the thread function are also given in the function parameters.
- The communication thread is created. The thread begins immediately to monitor the data queues for possible requests from the worker threads and for arriving messages from the network.

Once the initialization is completed, the worker threads and communication threads are running. The main program (the one that initialized the MPIT environment) can participate in the computation or it can call try to finalize the MPIT environment. In the latter case, the execution of the main thread is halted until all worker threads and the communication thread have terminated. By calling the finalization function, the main thread is pre-empted in favor of the communication thread. Therefore, it is beneficial that the main thread calls the finalization function and waits for all other threads to terminate. For optimal performance, the number of worker threads created should be one less than the number of processors in the workstation. The last processor is shared by the MPI main program and the communication thread.

3.3 Thread programming

During the configuration of the MPIT environment the user is able to specify the function executed by all threads immediately after their creation. MPIT also provides routines to control the execution of all threads within a workstation. There are two barriers that can be used to synchronize the execution of the threads. The first barrier is for the synchronization of all threads including the main thread, whereas the second barrier is for the synchronization of the worker threads. Furthermore, there are two routines that can be used to control the execution of a particular thread. It is possible to suspend the execution of a particular thread, and later resume its execution.

Without the above-mentioned enhancements, the thread programming model is identical to POSIX threads; all *pthread* routines work normally [11]. It should be noted that if threads share global memory locations, it is up to the user to implement synchronization methods to protect the memory locations at least from concurrent write operations. This requirement will be removed in the next version of MPIT where a method of controlling the shared data is provided.

3.4 Data transfer

MPIT's set of send and receive commands is similar to those of the MPI library. One of the goals of MPIT was to provide an identical interface with MPI and to hide all the complexity from the user. For example, the two most common commands, MPI_Send and MPI_Recv, have exactly the same syntax.

Perhaps, the most noticeable difference between MPIT and MPI is the way MPIT_Send function call operates. In the MPIT, the MPIT_Send function always returns immediately allowing the communication and computation to overlap. The only action the MPIT_Send function takes is to queue the data to the Data-Out queue. It is the responsibility of the communication thread to actually send the data to the source process. Due to this somewhat changed functionality of the MPIT_Send function, the MPIT implementation does not currently support any other kind of send functions.

Depending on the type of the receive call the following situations may occur during the reception of a message from a network: (1) If a non-blocking receive operation is performed (MPIT_Irecv), the Data-In queue is searched for a message that matches the parameters given in the function call. These parameters are the data type, source processor rank, and tag. If no match is found, the function returns with an error code. (2) If a blocking receive operation is performed (MPIT_Recv), the same search operation is carried out. However, if no matching message is found, the function does not return, but waits until a message with the matching parameters is entered into the queue. The queue is searched each time a new message is queued to see whether a match is found. Meanwhile, the halted thread is pre-empted to allow other threads to run, if any exist. (3) In MPIT it is possible to issue a receive command without specifying any parameters (MPIT_RecvAny). In this case, a message is returned to

the calling processor, if there is any kind of message in the Data-In queue for the processor. In addition to the returning of the data, all the function parameters are updated by the receive operation. Thus, the receiving process is informed about the type of data, the size of the data, source processor rank, and tag. This function call is very useful in the sense that the processors do not have to have prior information about the messages they are receiving.

3.5 Termination of the environment

The main thread should call the MPIT_Finalize() function to terminate the MPIT environment. The worker or communication threads can also call this function, but since they do not any effect on the environment, only the main thread can terminate the environment. The following operations take place in the function:

- The worker threads are joined. The execution of the main thread is halted, if no threads can be joined.
- The communication thread is joined. The MPIT_Finalize function guarantees that the communication thread is joined only after all worker threads have terminated and all their requests have been served.
- All memory allocated for the MPIT environment is released.

Once the MPIT environment is terminated the execution of the application can continue as a normal MPI application. However, it can reinitialize the MPIT environment after the termination of the environment either with the same or different parameters.

4 Results

There are a number of theoretical and practical benefits of MPIT. First, with a specific communication thread, worker threads are able to continue their work without any interruptions due to communication delays; the worker threads do not have to wait until a send operation is complete. Furthermore, the receive operations from the network are performed by the communication thread. The worker threads need only to issue a receive command that returns data to them from the Data-In queue. This reduces the time the worker threads have to spend on communication in general. In addition, MPIT allows receive operations that can receive any number of any kind of data from any process with any tag. This gives the user more flexibility with the communication, since there is no need to know the data size, type, and the source of the data prior to calling the receive function.

Second, MPIT optimizes the utilization of the resources indicated by the user in a workstation. The user does not have to employ all the available processors, and the user is even capable of terminating a number of threads during the parallel computation phase. The latter case is beneficial in a situation where the load of a

machine has increased, and all the threads are not receiving an equal amount of processor time.

Third, MPIT supports heterogeneous environments. The user can specify any number of threads for any workstation. By controlling the number of workstations the user is able to fully utilize the resources of all workstations participating in the computation. In addition, if automatic load balancing is enabled, the relative power indicators indicate the computing power of a workstation in proportion to the most or least powerful workstation. With the relative power information the automatic load balancing can perform optimal load balancing operations as requested by the threads.

Fourth, MPIT hides the complexity of thread management and communication. The programming paradigm does not change significantly; if the user is familiar with MPI programming, practically no new commands need to be learnt.

The performance of MPIT was tested with a special application written for the purpose of measuring the performance of MPIT. In the context of the test application, the term *process* refers to a workstation and each process can have a number of threads in it. The application consists of a master processor and a number of slave processors. The master process sends workloads to slave processes. There are two kinds of workunits in a workload received by a slave process: local workunits are processed by the slave process, whereas remote workunits are sent to processes that are responsible for processing them. The determination is based on the content of the workunit. The slave process goes through the workload and processes workunits assigned to it, and sends workunits not intended for it to other processes. The workload is processed sequentially and processing of a local workunit takes place immediately when one is found. In turn, remote workunits are sent to other processes when they are encountered in the workload. Thus, they are not sent out as a single message (containing all remote workunits) prior to the processing of the local workunits. This implementation simulates a situation where there are dependencies among the workunits and processes. In case a process has multiple threads in it, the threads process the workload so that each thread gets an equal number of workunits (regardless of whether the workunits are local or remote).

Initial tests have indicated that MPIT is capable of utilizing the resources in the network of workstations better than MPI. Table 1 shows the initial results of a comparison of MPIT and MPI. The results from 1 to 3 processors were achieved running the application in a 4-processor workstation utilizing three threads. The results from the test where one processor is used gives an indication of how long it takes to set up the MPIT environment. Setting up the MPIT environment takes approximately one second on a Pentium III 750MHz workstation running the Linux operating system. Once the number of processors is increased the more efficient utilization of resources in MPIT becomes evident. In the case where 4 processors are used (a total of 2 workstations: one with 3 processors and one with one processor), the performance improvement of MPIT over MPI is 41%. The improvement decreases as the amount of work per processor decreases, which can be seen in the test where 5 processors are used.

Table 1. Initial results from a comparison of MPIT and MPI

Number of processors	MPIT	MPI	Improvement
1	45.10	44.04	-2%
2	30.03	33.04	+10%
3	22.03	30.03	+36%
4	17.02	24.00	+41%
5	15.10	19.01	+26%

The power of MPIT can be seen with the test application. The test application benefits from MPIT's capabilities to use threads instead of processes on a workstation, and to minimize the need for communication by allocating a single process per workstation.

5 Conclusions

A new programming paradigm for efficient parallel computing on a cluster of SMP workstations was presented. MPIT (MPI and Threads) utilizes the MPI message passing library in communication among the workstations and threads to execute the code on the processors of a workstation. MPIT hides the thread creation and maintenance from the user. The interface of MPIT is very similar to the interface of the MPI library in that almost all of the same commands are available in MPIT and MPI.

MPIT utilizes threads instead of processes to take advantage of the resources of a workstation. The number of threads created in each workstation is controlled by the user. More threads can be created or the existing threads can be terminated during parallel computation. The use of worker threads in conjunction with a separate communication allows the worker threads to focus solely on processing their work, whereas the communication threads handle all communication operations. The communication thread sends data to other workstations and receives data from the workstations. The worker threads simply pass the data to be transmitted to the communication thread, and are then able to continue their execution. Furthermore, the reception of data by a worker thread includes a local function call to retrieve the data from a queue of the communication thread.

In addition to the features mentioned above, MPIT can perform load balancing of a data parallel application automatically. The load balancing operations are triggered by the user with a handful of function calls. MPIT hides the implementation including the communication from the user. The load balancing utilizes a client-server model, where a workstation is dedicated to act as a master for load balancing.

MPIT has a number of advantages. First of all, with the creation of the communication thread, worker threads do not have to perform any communication operations, except queue the data to the communication thread. Therefore, by overlapping computation and communication the performance of the parallel application is increased. Second, threads are able to utilize the system resources more efficiently than an equal number of processes. The context switch takes less time, and the threads automatically share a global memory allocated for the process in which they are created. The number of threads can be dynamically controlled by the user based on load situation in the workstation. Third, the MPIT implementation is hidden from the user. The user needs only to initialize the environment, and provide the code executed by the threads. The communication and thread management is handled by MPIT without user interaction.

A need for libraries such as MPIT has risen during the past years due to the increased number of multiprocessor-based workstation clusters. New and efficient programming paradigms are required to fully utilize available resources and minimize the overhead introduced by parallel execution. MPIT offers a programming paradigm that fulfills these requirements and can be adapted by any user with relative ease.

References

1. TOP500 Supercomputer Sites, University of Mannheim and University of Tennessee, http://www.top500.org.

2. MPI Forum: MPI-1.1 Standard. http://www.mpi-forum.org (1995).

3. MPI Forum: MPI-2 Standard. http://www.mpi-forum.org (1997).

4. Sunderam V.S.: PVM: A Framework for Parallel Distributed Computing. Concurrency: Theory and Practice **2** (1990) 315-339.

5. Tang, H., Shen, K., Yang, T.: Compile/Run-time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines. Proceedings of ACM Programming Principles of Parallel Processing (1999) 107-118.

6. Protopopov, B.V., Skjellum, A.: A multi-threaded Message Passing Interface (MPI) architecture: performance and program issues. Journal of Parallel and Distributed Computing **61** (2001) 449-466.

7. Haines, M., Cronk, D., Mehrotra, P.: On the Design of Chant: A Talking Threads Package. Proceedings of the Conference on Supercomputing (1994) 350-359.

8. Tanaka, Y., Matsuda, M., Kubota, K., Sato, M.: COMPaS: A Pentium Pro PC-Based SMP Cluster. High Performance Cluster Computing, Volume 1. 1st edition. Prentice Hall (1999) 661-681.

9. Parab, N., and Raghvedran, M.: Active Messages. High Performance Cluster Computing, Volume 1. 1st edition. Prentice Hall (1999) 270-300.

10. Bader, D.A., Jájá, J.: SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs). Journal of Parallel and Distributed Computing **58** (1999) 92-108.

11. Butenhof, D.R. Programming with POSIX Threads. Addison-Wesley, 1997, ISBN 0-201-64492-2.

Publication 6

Porras J., Huttunen P., Ikonen J.: Accelerating Ray Tracing Based Cellular Radio Coverage Calculation by Parallel Computing Techniques. Annual Review of Communications, Vol. 53, 2000.

Accelerating ray tracing based cellular radio coverage calculation by parallel computing techniques

Jari Porras, Pentti Huttunen and Jouni Ikonen
Lappeenranta University of Technology, P.O. Box 20, FIN-53851 Lappeenranta, Finland
{Jari.Porras, Pentti.Huttunen, Jouni.Ikonen}@lut.fi

INTRODUCTION

Calculation of the radio wave propagation and the field strength is one of the most critical tasks in any computer based cellular network planning system or simulator. Because of the increasing demand for the capacity in cities the cells are getting smaller and smaller and there is a need for an accurate prediction of coverage in a complex urban geometrical setting. Especially rapid growth of the GSM phone population has resulted a large number of microcell simulators [Hei96, Pap98, Raj96a, Sip96]. Several techniques and models have been used to simulate radio wave propagation in all kind of environments, i.e. indoor [Ina94, Laf90, Läh93, Raj96a, Sei92] and outdoor [Bag95, Gre94, Hei96, Hut98, Pap98, Tut96]. Some of the models are empirical, or statistical based on the field strength measurements and experience, see e.g. [Par92], and some of them are deterministic using accurate maps of the service area as an input.

Recently ray tracing [Fri95, Hut98, Ina94, Sal94; Sip96, Tut96] method has become very popular technique to be used in the propagation model simulation. In [Lia97] there is a good overview of the ray tracing technique. Ray tracing based simulations are used for finding the best possible locations for transmitters before actually installing them. Ray tracing enables accurate results but it also demands a huge amount of calculations. Due to the high amount of calculations the use of ray tracing requires a powerful computer or a lot of time. As several simulations need to be run before optimal locations can be found the execution time of the simulation must be kept reasonable. Improved algorithms have been developed [Hei96, Tut96] but the simulation time remains too long for the iterative process of radio network planning.

Parallel computing offers a solution for the time consuming coverage calculations. Calculation times can be significantly reduced by partitioning the problem into independent parts and by executing the parts in parallel. Coverage calculation is a suitable application for parallel execution as the independent parts can be easily found. However, the size of the different parts may vary which may result in uneven execution times. In order to achieve the best possible speedup some kind of a work balancing algorithm need to be implemented.

PROPAGATION MODELING

The idea of the propagation modeling is to simulate radio wave propagation in urban and rural environments. Propagation models can be used for route and coverage calculations. Route calculation is used for the computation of field strengths on a

mobile receiver's path. The field strength information received from the route calculation indicates how the received signal changes when the mobile is moving on the simulation area. In coverage calculation the field strengths are calculated for a certain area. This area is divided into receiving points which will have the field strength information calculated at the end of the simulation. Through the coverage calculation the true coverage of the transmitter can be determined.

In the coverage calculation the field strength needs to be calculated for every receiving point. For simulation purposes it is sufficient to create a receiving grid over the simulation area. Each square of the receiving grid represents one receiving point. Density of the grid may vary according to the desired accuracy. A 4*4 meter square is usually considered accurate enough. Figure1 presents an example of the receiving grid on the given area.

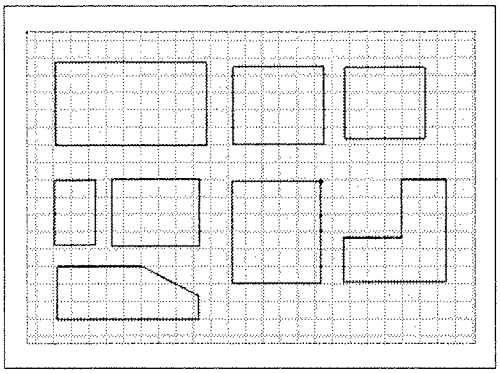


Figure 1. Receiving points represented by a grid in the simulation area.

Signal loses its power as it propagates from the transmitter to the receiving point [Feu94, Hat80, Laf90, Läh93, Oku68, Pap98, Sei92]. Different elements affect to the received signal power. The propagation modeling is divided into three components:

- Distance
- Reflections
- Diffraction

Signal loses its power according to the distance it propagates before it reaches the receiving point. In an empty area the signal suffers only from the free space loss. Free space loss depends on the distance between the transmitter and the receiving point as well as the frequency used. Obstacles between the transmitter and the receiving point will further degrade the signal. Obstacles also cause the signal to reflect and diffract. If the signal intersects an obstacle a new, reflected signal is created. Reflected signal reflects from the obstacle whereas the originally transmitted signal goes through the obstacle. Power loss of the signal depends on the material of the obstacle. Diffraction happens when the signal arrives to a corner of an obstacle. In the corner the arrived signal disperses to all directions. This will create a great deal of new signals, which increases the amount of calculation needed in the propagation modeling.

Different propagation models can be categorized into two groups according to the use of dimensions. In 2D modeling only horizontal plane is considered. These models

may be sufficient enough if the studied area is adequately flat. However, in the real environment, e.g. cities, the terrain and its changes have an effect to the signal propagation. In these cases vertical plane should also be considered. This 3D modeling complicates the calculation of signal propagation and thus increases the demand for computing power.

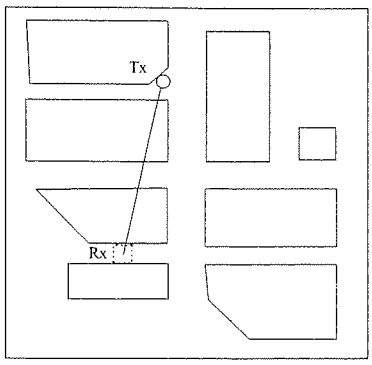


Figure 2. Point-to-point propagation model.

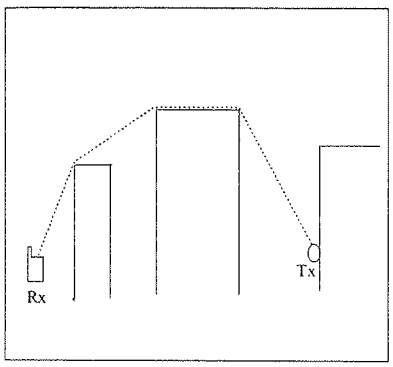


Figure 3. Diffraction over the obstacles in 3D model.

Two basic methods have been proposed for the propagation modeling: point-to-point and ray tracing. Point-to-point propagation considers only the straight line from the transmitter to the receiver. This type of modeling is illustrated in Figure2. In 2D point-to-point propagation modeling the received signal strength is affected only by the distance and the signal attenuation due to the obstacles on the signal path. In 3D model the diffraction over the obstacles can be considered. This is illustrated in Figure3. Point-to-point method is simple to implement and it does not require huge amount of calculations. Because reflections and (horizontal) diffraction are ignored the accuracy of this method is poor. Only rough estimates can be achieved by using this method.

Ray based propagation modeling

In many recently implemented propagation model simulators the propagation is based on a ray tracing method. In a ray based model the transmitter sends signal rays to all directions unlike in the point-to-point method, where only a single ray is inspected. Each ray is traced until the power of it drops below a given threshold value. This method is more accurate than point-to-point method since reflections and diffractions are included (see Figure4). As reflections and diffractions generate multiple new rays the number of rays increase exponentially. Therefore, ray based models are more complex than point-to-point models and they require more calculations. The angle between the launched rays must be small enough to achieve good results. As the angle is decreased the amount of calculation is increased. The extent calculation and the problem of tracking arrived rays to receiving point make the use of ray based models quite complex. The amount of calculation increases enormously if there are several receiving points, i.e. a receiving grid.

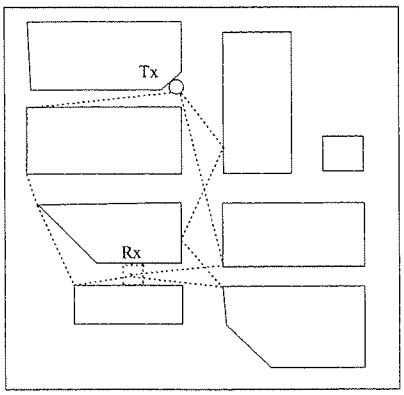


Figure 4. Ray tracing based propagation model.

Ray tracing in our coverage calculation

Due to the problems in the basic ray tracing method a more sophisticated and accurate method should be used. A more powerful ray tracing method is based on the use of transmitting points' line-of-sight (LOS) polygons [Hei96]. A transmitting point's LOS polygon represents the area seen from the transmitting point. This area illustrates those receiving points where the arriving ray would have diffracted. The field strength values of the receiving points in this area are updated according to the power value of rays arriving to the transmitting point. LOS polygons are created for the transmitter as well as to all diffracting corners. Figure5 presents a LOS polygon of a diffracting corner C. Due to the use of LOS polygons single rays need not to be traced.

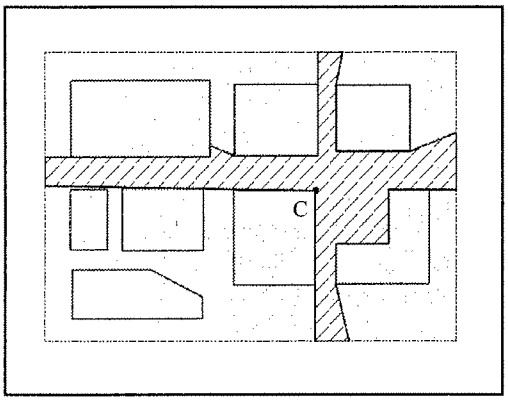


Figure 5. Line-of-sight polygon of corner C.

PARALLEL APPROACH TO OUR RAY BASED PROPAGATION MODEL

In coverage calculation most of the time is consumed in updating the field strength values of the receiving points. Both vertical and horizontal plane calculations are suitable for the parallelization. In the vertical plane calculation where the diffraction over the obstacles, e.g. buildings, is considered, the division between processors is trivial, since each receiving point will take approximately the same time to process.

Therefore, a static and equal division of receiving points to the processing elements is usually sufficient enough.

In horizontal plane calculation the field strength values are updated by filling the LOS polygons. Since the polygons are independent objects they can be processed in parallel. However, the sizes of LOS polygons are not equal as the area seen from corners is different. The computation time of a single LOS polygon depends on the number of its receiving points and the number of times the polygon is filled. A LOS polygon is filled as many times as there are rays arriving to the corner. This makes the work balancing more challenging. Therefore, the static division method used in vertical plane calculation is not sufficient in the horizontal plane calculation. Several alternatives for the work balancing were studied.

- In *Corner Division* algorithm the number of diffracting corners is divided equally for the processing elements. Last processing element has fewer corners than the others. As the work required for different corners, i.e. points within a LOS polygon, may differ considerably this method is not the best work-balancing algorithm. If the work is sufficiently equal among the LOS polygons then this method may be considered.
- In *Work Division* algorithm the work is allocated for processing elements (PE) according to the pre-computed average of the work in LOS polygons. Corners are added for the PE as long as the work allocated for the PE exceeds the average. As no sorting is done for the corners this simple method of dividing the work is quite fast even with high number of corners. This method works well if the LOS polygons are sufficiently equal in sizes but may fail if the work of polygons differs considerably.
- The *WorkPool* algorithm makes the division of corners among processors according to the real amount of work required by the corner. In order to realize the real amount of work the total number of receiving points per LOS polygon is first calculated and then multiplied by the number of rays arriving to the base corner of the LOS polygon. The real work of a LOS polygon is stored with the number of corner into a table. This table is then sorted in descending order according to the real amount of work. LOS polygons are processed starting from the largest, i.e. from the top of the table. The table acts like a pool from where processors fetch a new LOS polygon to process until all polygons are filled. A free processor will always process the largest free polygon. The WorkPool algorithm proved to be a suitable solution for the coverage calculation.

3D ray based propagation model consist of several elements that can be parallelized quite efficiently. The following paragraphs present those elements that were considered in our study:

Over-the-roofs diffraction

In the over-the-roofs diffraction the received power is calculated to each outdoor receiving point by calculating the distance loss and the diffraction loss over the buildings between the transmitter and the receiver. Since each receiving point takes about the same time to calculate the division between processors may use simple

methods: The number of receiving points may be divided by the number of PEs. Each PE will then process the given amount of receiving points.

Multiple reflections

Parallel implementation of reflections is based on division of the walls seen by the transmitter into PEs. PEs will recursively calculate the reflections. Because the reflection polygons (see e.g. [Hei96]) are often overlapping, some kind of a method restricting simultaneous updates need to be implemented. The simple division of walls seen from transmitter to PEs does not provide optimal results because the calculation load is not divided evenly among the PEs. However, the execution time of reflections is only a small part of the total execution time.

Multiple diffractions

Calculation of multiple diffractions is a prerequisite for a micro cellular ray tracing program to work accurately in the shadow areas (or otherwise complex combinations of reflection, diffraction and scattering are needed). Since multiple diffractions are the worst computational bottleneck of the ray tracing based propagation models, the load sharing among processors need to be designed carefully to get the best parallel speed-ups. In the calculation of multiple diffractions most of the time is spent in filling the LOS-polygons for each corner. The basic idea in the parallel solution is to divide LOS-polygons among the PEs. As in the reflection calculation the LOS polygons are often overlapping, the simultaneous update of receive values need to be restricted. Since the areas of LOS polygons are not equal some kind of a work balancing algorithm is needed. The WorkPool algorithm seemed to work well with the diffrections.

Indoor coverage

Indoor coverage model is based on the "doughnut way of thinking". This means roughly that after the field strengths around buildings has been calculated by the outdoor model a separate indoor model determines the field strengths inside the buildings by using the results around the buildings as input. For details, see [Raj96b]. In this project only the COST231 penetration model was implemented. Each wall of a building contributes to a receiving point inside the building according to field strength outside the building, the distance from the wall and attenuation parameters. The model was implemented by creating "an orthogonal LOS polygon" for each wall of a building under processing. This polygon includes those receiving points inside the building, which can be connected orthogonally to the contributing wall by a straight line, which does not intersect other walls of the building.

In the parallel solution the buildings are simply divided into PEs. After a PE has processed a building it picks up the next building not yet processed from a pool of buildings. Thus, the buildings have here the same role as the LOS-polygons in the case of multiple diffractions. However, the buildings are not sorted here according to estimated work amount like the LOS polygons are in the calculation of multiple diffractions. This means that the load is not balanced optimally.

EXECUTION ENVIRONMENTS

Simulation experiments were performed by using shared and distributed memory environments. Both types of parallel environment were used to find out the properties of the application. Shared memory environment is quite easy to program as processors can access the same memory and threads can be used efficiently. However the number of available processors is usually limited to quite a small number. In distributed memory environment message passing must be used. This is generally slower than the use of shared memory, but more processors are usually available.

Simulation experiments in shared memory environment were run by using a 4 processor Sun workstation and a Digital AlphaServer8400 with 10 Alpha processors. The implementations were based on the use of threads. Simulation experiments in distributed memory environment were run by using Cray T3E, which had 224 application processors. The distributed implementation was based on the use of MPI (Message Passing Interface) and SHMEM (Shared Memory) libraries.

TEST CASES AND RESULTS

Five test cases are used for the evaluation of the parallel coverage calculation. Test cases are from two kind of cities, European (maps 1-3) and Asian (maps 4 and 5). In all cases the simulation area represents an area covered by a microcell network. Maps 1-3 present a typical European city with a lot of small buildings and empty space, i.e. parks and streets. The streets are wide, long and straight. The amount of free space is relatively high compared to the whole area. Maps 4 and 5 are from an Asian city. They have large buildings with only little free space. The streets are narrower and shorter than in maps 1-3. Test cases and their parameters are presented in Table 3

Table 3. Test cases used in the experiments.

Test case	Size of the calculation area (m)	Number of receiving points to update	Number of corners
Map1	1 640 x 1 470	84140	1436
Map2	1500 x 1380	71258	1251
Map3	1250 x 1730	96278	999
Map4	1270 x 840	46656	511
Map5	1270 x 840	46656	511

In each simulation experiment the coverage is calculated only for a single transmitter. Spacing of the receiving grid in all test cases is 4 meters, i.e. each receiving point represents a square of 4*4 meters. Number of receiving points to update -column tells the amount of outdoor receiving points, which will have the field strength information at the end of the simulation. Actual number of updates is bigger because each receiving point is likely to be updated several times. Number of corners -column indicates the actual number of diffracting corners in the simulation area. Each map has been chosen so that it represents different characteristics of cities. Some examples of these maps can be seen in appendix 1.

The following subsections present the simulation results of different maps in the specified environments. The experiments in the 4 processor Sun workstation are analyzed quite thoroughly whereas other environments are not considered so deeply.

4 processor Sun workstation

Figure6 shows the total execution times in seconds for the maps 1 - 5. It can be notices that execution times decrease nicely as the number of processors is increased. The effect of characteristics of different maps can be observed. The number of receiving points outside the building seems to have a clear effect to the execution time. Results from maps 4 and 5 show that the placement of the base station has only a small effect to the execution time.

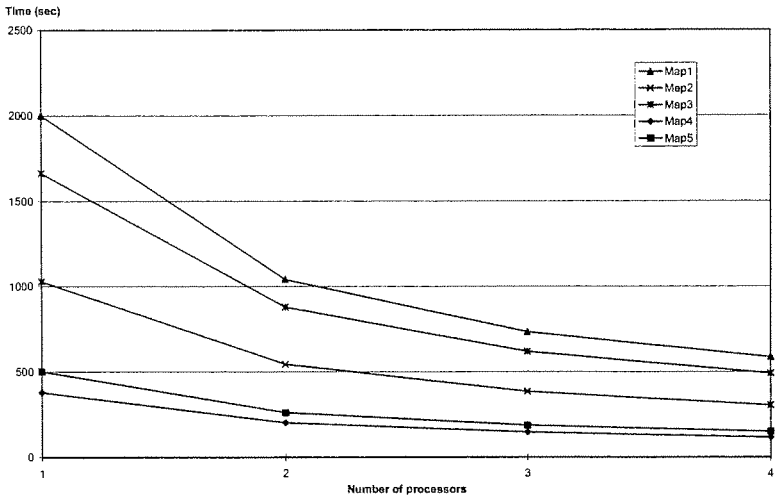


Figure 6. Total execution times as a function of number of processors.

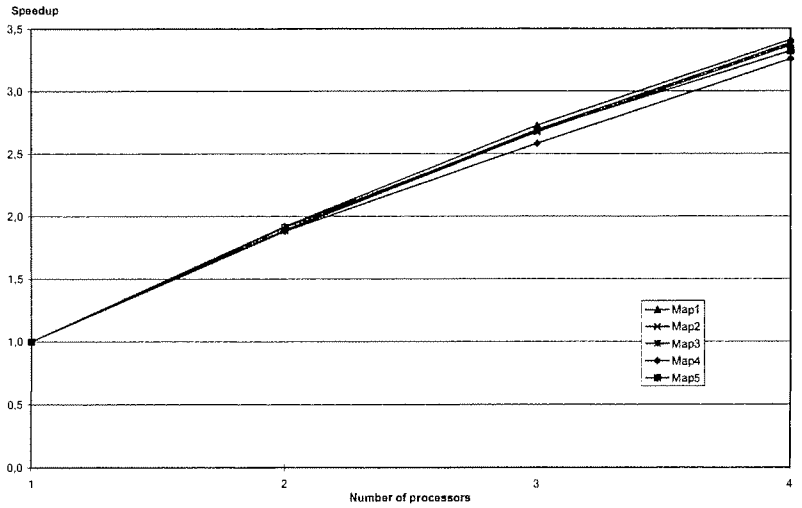


Figure 7. Speedups compared to the single processor execution.

Figure7 shows the speedups for the same test cases. It can be seen that full linear reduction of execution time was not achieved. This was mostly due to the sequential parts of the code from which disk I/O is the most significant reason. It can be seen from Figure7 that scenarios with a great number of receiving points and buildings in the computation area will gain slightly more of the parallel processing.

Table 4 shows the execution times of each distinctive task of the ray based coverage calculation. Execution times are presented for one to four processors. Thus the effect of parallelization in different tasks can be observed. Last column of the table indicates the percentage of processors allocated for the simulation; 400% would mean that all 4 processors have been involved with the calculation during the whole time. It can be noticed that multidiffraction is the dominant part of the propagation modeling. It can also be noticed the parallelization works pretty well in multidiffraction task.

Table 4. Run times for individual tasks in seconds

Testcase	Nunber of processors	Total execution time	Sequential part	Over diffraction	Reflections	Multi diffractions	Indoor propagation	% of processors max. 400%
Map1	1	2000,71	40,35	41,63	33,28	1880,44	5,01	99
Map1	2	1041,67	39,02	23,13	11,89	964,87	2,76	196
Map1	3	733,93	39,06	18,96	12,06	661,84	2,01	284
Map1	4	586,65	40,78	16,47	12,12	515,44	1,84	363
Map2	1	1049,46	32,93	36,67	3,80	971,53	4,53	99
Map2	2	545,07	32,51	19,98	1,86	488,34	2,38	192
Map2	3	384,81	34,17	16,13	1,96	330,24	2,31	275
Map2	4	305,60	33,55	13,04	1,96	255,44	1,61	351
Map3	1	1716,76	34,12	52,94	31,19	1595,55	2,96	99
Map3	2	878,86	33,68	30,84	10,85	801,92	1,57	195
Map3	3	618,48	34,32	25,94	9,67	547,33	1,22	282
Map3	4	491,81	34,13	21,21	9,76	425,58	1,13	361
Map4	1	379,20	15,21	28,17	15,13	319,20	1,49	99
Map4	2	201,52	14,96	18,28	7,17	160,21	0,90	187
Map4	3	146,90	15,67	14,05	7,38	109,20	0,60	260
Map4	4	116,44	14,92	11,21	7,18	82,54	0,59	329
Map5	1	501,59	15,26	22,37	11,16	451,30	1,50	99
Map5	2	262,15	15,03	13,46	5,63	227,23	0,80	191
Map5	3	186,93	14,99	11,01	5,41	154,91	0,61	272
Map5	4	151,11	15,02	10,33	5,65	119,52	0,59	342

Table 5. Speedups for the individual tasks.

Testcase	Number of processors	Total execution time	Over diffraction	Reflections	Multi diffractions	Indoor propagation
Map1	1	1,00	1,00	1,00	1,00	1,00
Map1	2	1,92	1,80	2,80	1,95	1,82
Map1	3	2,73	2,20	2,76	2,84	2,49
Map1	4	3,41	2,53	2,75	3,65	2,72
Map2	1	1,00	1,00	1,00	1,00	1,00
Map2	2	1,93	1,84	2,04	1,99	1,90
Map2	3	2,73	2,27	1,94	2,94	1,96
Map2	4	3,43	2,81	1,94	3,80	2,81
Map3	1	1,00	1,00	1,00	1,00	1,00
Map3	2	1,95	1,72	2,87	1,99	1,89
Map3	3	2,78	2,04	3,23	2,92	2,43
Map3	4	3,49	2,50	3,20	3,75	2,62
Map4	1	1,00	1,00	1,00	1,00	1,00
Map4	2	1,88	1,54	2,11	1,99	1,66
Map4	3	2,58	2,00	2,05	2,92	2,48
Map4	4	3,26	2,51	2,11	3,87	2,53
Map5	1	1,00	1,00	1,00	1,00	1,00
Map5	2	1,91	1,66	1,98	1,99	1,88
Map5	3	2,68	2,03	2,06	2,91	2,46
Map5	4	3,32	2,17	1,98	3,78	2,54

Speedups of individual tasks are presented in Table 5. It can be observed that the best parallel gain is achieved with multiple diffractions in which the load balancing is performed most carefully and which is the most time consuming part of the calculation. With 4 processors the maximum speedup achieved is 3.87. In some cases multiple reflections achieve speedup over 3. Other parts show only moderate speedups. One reason for this is that in the other parts there is a smaller amount of work, which means higher relative overhead of thread handling. Also the lack of efficient load balancing in the other parts affects to the achieved results.

AlphaServer8400

Figure8 presents the speedups achieved in 10 processor AlphaServer8400 as a function of the number of threads. Parallel execution time is compared to the simulation time with only one thread. Simulations were performed with 1-10 processors. The number of threads was varied between 1 and 14 in order to find out the effect of additional threads. It can be observed that the simulation achieves best results with 10 threads and thus no additional threads are necessary. Optimal speedup cannot be achieved as the simulator contains sequential parts. With 10 threads the speedups vary from 5,14 to 6,12. The difference between speedups can be explained with characteristics of each map. The best speedups are achieved with maps that contain a lot of receiving points. Due to the sequential part the achievable speedups are moderate.

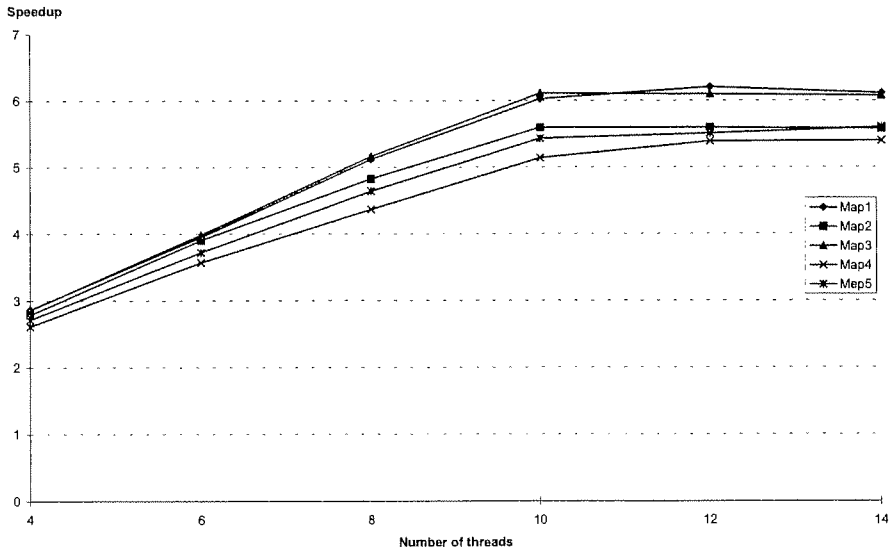


Figure 8. Speedups as a function of the number of threads in AplhaServer environment.

Cray T3E

Figure9 presents the speedups achieved in distributed environment as a function of the number of processors. Parallel simulation times are compared to the simulation time of a single processor. Although 224 processors were available in the environment only 32 was used at most. Again test cases perform quite similarly with small number of processors. With 32 processors the difference between different test cases is significant. It can be noticed that good speedups (8,7 - 14,1 with 32 processors) can be achieved but with large number of processors the sequential part and the communication will eventually start dominating the execution time.

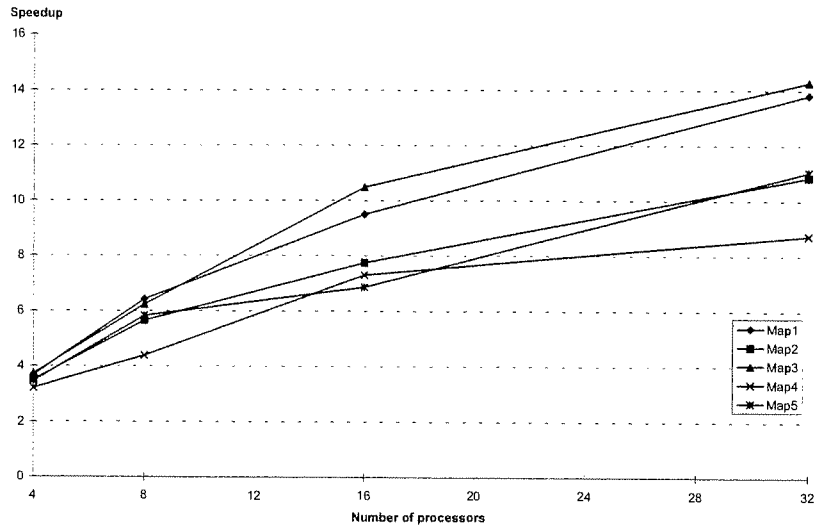


Figure 9. Speedups as a function of processors in Cray T3E environment.

CONCLUSIONS

The main target in the research presented in this paper is to reduce the execution times of a ray tracing based coverage calculation by using parallel processing techniques. It can be seen from the test results that the run times are decreased significantly in a shared memory system though fully linear speedups are not achieved. The best speedup achieved with 4-processor environment is 3.4 and with the 10-processor environment little over 6 which can be considered as good results. Linear speedups can not be easily achieved since there will always be a sequential part. Another factor, which limits the speedups, is the overhead generated by the parallel execution.

Implementation of the parallel versions presented in this paper was done in a shared memory environment by using threads and in distributed environment by using message-passing primitives. With threads the division of work among the processors is simple to carry out. When the thread calls were added to the existing sequential code only some small changes were required. Only few thread calls were needed to make the model run in parallel. Synchronization between the threads was the most time consuming part of the coding work, as mutex lock was needed. However they

cannot be avoided if correct results are required. It is very easy to produce miserable bugs with threads if it is not made sure that different threads do not write to the same memory addresses simultaneously or use the same memory addresses in a wrong way, not apparent in the sequential version. The use of message passing in distributed environment was much harder than the use of threads. Structure of the parallel execution was changes for efficient distributed execution of the propagation model in distributed environment. Due to the relative small size of the problem and cost of programming work and calculation, a massively parallel solution seemed not to be an applicable alternative.

ACKNOWLEDGEMENT

Authors would like to thank Kari Sipilä from Nokia Telecommunication for the sequential simulator and co-operation through the parallelization process. Authors would also like to thank Center for Scientific Computing Finland for the possibility to use the high performance parallel computers.

REFERENCES

[Bag95] Bagrodia R., Gerla M., Leinrock L., Short J. and Tsai T.: A Hierarchical Simulation Environment for Mobile Wireless Networks, *Proceedings of the 1995 Winter Simulation Conference*, 1995.

[But97] Butenhof D.: *Programming with POSIX Threads*, Addison Wesley, 1997.

[Cra96a] Cray Research: *Message Passing Toolkit: MPI programmer's manual*, Cray Research Inc., 1996.

[Cra96b] Cray Research: *CRAY T3E Applications Programming*, Cray Research Inc., 1996.

[Cra97] Cray Research: *Cray T3E Optimization*, Cray Research Inc., 1997.

[Feu94] Feuerstein, M. et. al.: Path loss, Delay spread, and Outage Models as Functions of Antenna Height for Microcellular System Design, *IEEE Transactions on vehicular technology*, Vol 43, NO. 3, August 1994

[Fri95] Fritsch T., Tutschku K., Leibnitz K.: Field strength prediction by ray-tracing for adaptive base station positioning in mobile communication networks, University of Wurzburg, *Research report No. 122*, Aug. 1995.

[Gre94] Greenberg A., Lubachevsky B., Nicol D. and Wright P.: Efficient Massively Parallel Simulation of Dynamic Channel Assignment Schemes for Wireless Cellular Communications, *Proceedings of the PADS'94*, 1994.

[Hat80] Hata M.: :Empirical Formula for propagation loss in land mobile radio services, *IEEE Transactions on vehicular technology*, Vol. VT-29, no. 3, 1980.

[Hei96] Heiska K. and Kangas. A.: Microcell Propagation Model for Network Planning, *Proceedings of the IEEE PIMRC 96* , 1996.

[Hut98] Huttunen P., Porras J., Ikonen J. and Sipilä K.: Using Cray T3E for the Parallel Simulation of Cellular Radio Coverage Calculation, *Proceedings of the Eurosim'98*, 1998.

[Ina94] Inanoglu H. and Topuz, E.: A ray based indoor propagation model for DECT applications, *European Simulation Symposium 1994*.

[Kle96] Kleiman S., Shah D. and Smaalders B.: *Programming with Threads*, Prentice Hall, 1996.

[Laf90] Lafortune J. and Lecours M.: Measurement and Modeling of Propagation Losses in a Building at 900 MHz, *IEEE Transactions on Vehicular Technology*, Vol. 39, No. 2, 1990.

[Lew96] Lewis B. and Berg D.: *A Guide to Multithreaded Programming*, Prentice Hall, 1996.

[Lia97] Liang G. and Bertoni H. L.: Review of ray modeling techniques for site specific propagation prediction, *Wireless Communications*, TDMA versus CDMA, ed. S.G. Glisic, Kluwer London, 1997.

[Läh93] Lähteenmäki J.: Determination of Dominant Signal Paths for Indoor Radio Channels at 1.7 GHz, *Proceedings of Personal Indoor Mobile Radio Conference*, Yokohama, 1993.

[Oku68] Okumura Y., Ohmori E., Kawano T. and Fukuda K.: Field Strength and Its Variability in VHF anf UHF Land-Mobile Radio Service, *Review of The Electrical Communication Laboratory*, Vol. 16, No. 9-10, 1968.

[Pap98] Papadakis N., Kanatas A. and Constantinou P.: Microcellular Propagation Measurements and Simulation at 1.8 GHz in Urban Radio Environment, *IEEE Transactions on Vehicular Technology*, Vol. 47, No. 3, 1998.

[Par92] Parsons J.D: *The Mobile Propagation Channel*, Wiley & Sons, Inc. New York, 1992.

[Raj96a] Rajala J., Sipilä K. and Heiska K.: Predicting In-Building Coverage for Microcells and Small Macrocells, *Nokia Research Report*, 1996.

[Raj96b] Rajala J. and Sipilä K.: uCell Model 3D extension for NPS/X, *Nokia Research Report*, 1996.

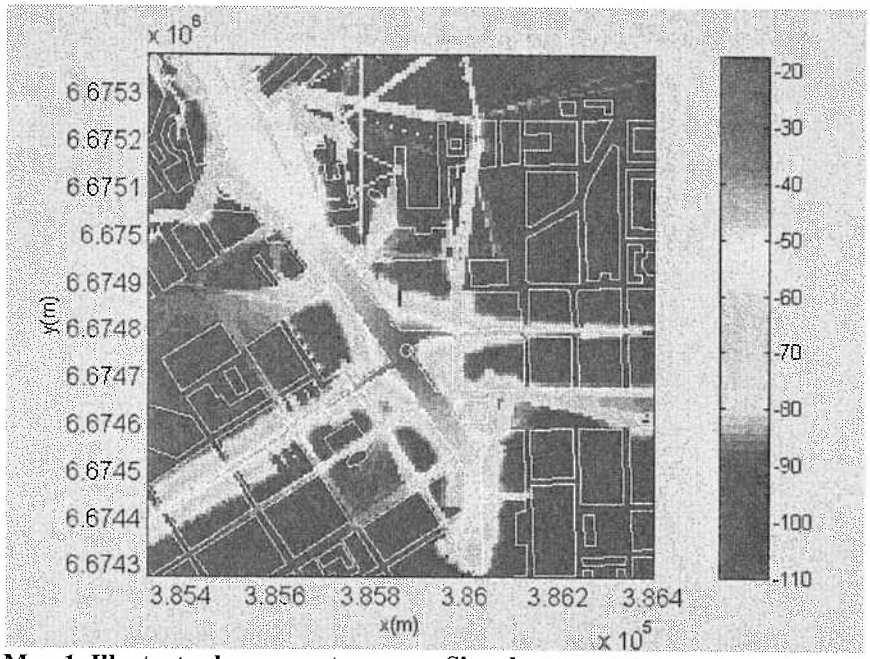
[Sal94] Salmi M.: Parallel ray tracing in propagation modeling of indoor mobile radio communication, *Research Report 51*, Department of Information Technology, Lappeenranta University of Technology, Lappeenranta, Finland, 1994.

[Sei92] Seidel S. and Rappaport T.: 914 MHz Path Loss Prediction Models for Indoor Wireless Communications in Multifloored Buildings, *IEEE Transactions on Antennas and Propagation*, Vol. 40, No. 2, 1992.

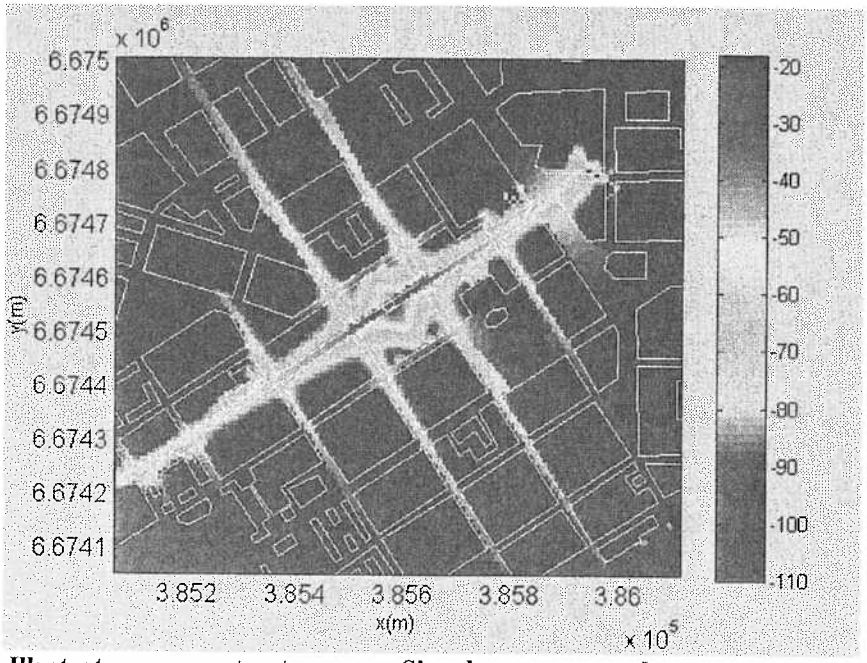
[Sip96] Sipilä K. and Heiska K.: Can ray tracing be used as a fading generator in simulating micro cellular mobile radio system?, *The 8th International Conference on Wireless Communications* , 1996.

[Tut96] Tutschku, K., Leibnitz, K.: Fast Ray-Tracing for Field Strength Prediction in Cellular Mobile Network Planning, University of Wurzburg, *Research report No. 134*, 1996.

Appendix 1



Map 1. Illustrates large empty spaces. Signal propagates to very large area.



Map2. Illustrates narrow street canyons. Signal propagates only to streets near the base station.

Publication 7

Huttunen P., Ikonen J., and Porras J.: Parallelization of a WCDMA System Simulator for a Shared Memory Multiprocessor Machine. In Proceedings of European Simulation Symposium, Erlangen-Nuremberg, Germany, October 26-28, 1999, pp. 556-560.

PARALLELIZATION OF A WCDMA SYSTEM SIMULATOR FOR A SHARED MEMORY MULTIPROCESSOR MACHINE

Pentti Huttunen, Jouni Ikonen, and Jari Porras
Lappeenranta University of Technology, P.O.Box 20, FIN-53851 Lappeenranta, Finland
{Pentti.Huttunen, Jari.Porras, Jouni.Ikonen}@lut.fi

KEYWORDS

3rd generation mobile networks, WCDMA, shared memory multiprocessors, work balancing

ABSTRACT

In this paper a parallelization process of a Wide-band Code Division Multiple Access (WCDMA) system simulator is presented. The WCDMA technique is considered as 3rd generation mobile standard for the air interface. The simulation process demands a great deal of calculation to be completed. The simulation times of the sequential simulator have been far too excessive for interactive use. Therefore, in this paper a parallel approach is presented to speedup the execution times. The achieved results indicate that with the correct use of work balancing algorithms promising results can be achieved. With the use of 4 processors the speedup of 3.1 was achieved. However, the total execution time was reduced to half due to the large sequential part still existing in the simulator.

INTRODUCTION

As a 3rd generation mobile standard, the WCDMA technique offers higher transfer rates than the presently used GSM. Due to the characteristics of the WCDMA technique, the simulation of WCDMA based networks is very different from the simulation of GSM systems. Major differences in power control algorithms, make the simulation process of the WCDMA system simulator considerably more complicated. In this paper a WCDMA system simulator is presented. The simulator is used as a platform for WCDMA studies as well as for algorithm design. This paper focuses on the parallelization process of the simulator. The two most time-consuming parts of the simulator were reprogrammed to take advantage of an SMP machine with 6 processors. In the following chapters, the WCDMA technique is considered more closely. Then the sequential WCDMA system simulator is explained. Finally, the parallelization of the system simulator is discussed in greater detail with the achieved results.

WCDMA TECHNIQUE

Wide-band Code Division Multiple Access is one of the 3rd generation mobile techniques /1/. The International Telecommunication Union (ITU) has not yet made a decision about the global standard for 3rd generation mobile networks (IMS-2000) /6/. The standardization organization of Europe, European Telecommunication Standardization Institute (ETSI), and Japan, Association of Radio Businesses (ARIB), are in favor of WCDMA whereas Telecommunication Industry Association (TIA), the corresponding organization in the USA, has made its own proposal based on CDMA-1 /20/. WCDMA and CDMA-1 techniques are very different. The simulator presented in this paper uses WCDMA as the air interface. In the next subsections, three main characteristics of the WCDMA technique are explored. A more comprehensive introduction about WCDMA can be found from /16, 17, 19, 23/.

Basics of WCDMA

Unlike the 2nd generation mobile techniques, WCDMA is based on neither FDMA nor TDMA /1, 3, 17/. Both of these techniques divide the available frequency or time slots into parts, which are assigned to users; if the data being sent is bursty or irregular, the efficiency of FDMA and TDMA is poor. In WCDMA the frequency is not divided into parts to be used by different users. Therefore, the transmitted signals are continuous, whereas in TDMA, the transmission is based on the time division of slots /8/. The whole spectrum, allocated for WCDMA, is at the disposal of all users on account of spread spectrum modulation /22/. The idea of spread spectrum modulation is to spread the transmitted signal over a wide frequency band, which is much wider than the minimum bandwidth required to transmit data.

Spreading of the signal

Spreading is based on a spreading code, which is a random sequence of bits. Each transmitter has a distinct spreading code, which it uses to spread the transmitted information and despread the received information. The spreading code of the transmitter has to be unique, otherwise, the transmitted information of multiple transmitters are mixed. The use of a spreading code to spread the signal over the spectrum causes the transmitted signal appears as noise. The transmission can also be detected as a slight increase in the interference level. However, only the receiver of the transmission, which knows the spreading code of the transmitter, is able to despread the signal and recover the actual data.

The allocation of spreading code is done dynamically when a call is established. First, a mobile station uses random access channels to indicate to the base station that it wants to establish a call. The spreading codes for the use of random access are pre-defined and, therefore, known by the mobile stations. After receiving the call establishment request from the mobile station, the Radio Network Controller (RNC) generates a random bit sequence, which is then assigned to the mobile station's spreading code for the call being established. When the spreading code is communicated to the mobile station, it starts to use the code to spread and despread the signal.

Power control

Since all mobile stations are able to use the whole spectrum to transmit the signal, power needs to be controlled /22/. The purpose of power control is to make sure that all base stations and mobile stations are able to transmit and receive signals without causing an excessive amount of interference to the network. The goal is to maintain equality in the powers of all mobile stations as much as possible. The power control algorithms are executed 1600 times a second, that is, every 0.625 ms.

The power control is an essential feature in the WCDMA technique, since it requires that received powers of all mobile stations be equal. The increase in the transmission power of a mobile station has a two-fold effect. The quality of the connection is improved. On the other hand, the increase in

power introduces more interference to other mobile stations, which then have to increase their transmission powers to overcome the increase in interference.

Each mobile station generates more interference to the network causing the other mobile stations to adjust their transmission powers. Therefore, the network has to be designed so that the maximum interference level does not exceed the maximum transmission power of the mobile stations. Consequently, the networks are usually designed so that their maximum acceptable load is about 50 % of the full capacity, since interference increases exponentially when the load grows.

Soft handover

In WCDMA a mobile station can be connected to multiple base stations at the same time, unlike in GSM, where each mobile stations is connected to one base station at a time. The use of multiple base stations in WCDMA is supported by the fact that the mobile station receives signals from the base stations, which enables the use of lower power levels and achieves better performance. When the mobile station is closing in on the border of a cell, the ability to be connected to the base station serving the current cell and to the base station serving the cell where the mobile station is about to move, is a major advantage. If the mobile station does not move across the border of the cell, it can stay connected to both base stations and, therefore, get better service. However, the mobile station prepares for a handover if the signal strength of the neighbor cell exceeds a certain threshold that it is still under the current base station's signal strength. When the need for handover occurs, the mobile station either drops the connection to the base station or establishes a new connection to a new base station. In the case where the connection is dropped, the mobile station has already connected to the base station serving the current cell, so, the handover is much smoother and requires less transmission power.

The above mentioned soft handover does not exist in 2nd generation mobile networks. For example, in the GSM technique, the handover is known as hard handover. In hard handover, the decision of whether handover is needed is based on comparison of the signal strengths of the current cell and the neighboring one. When the signal strength of the neighbor cell exceeds the signal strength of the current cell with a threshold, the handover is performed. Therefore, the mobile station is connected to only one base station at a time; this causes the handover to demand more power. In addition, there is an interruption in the transmission while the handover takes place. In general, hard handover is not as efficient as the soft handover.

WCDMA SIMULATOR AND THE PARALLELIZATION PROCESS

The simulator is used as a test environment for WCDMA studies and algorithm design. Figure1 depicts the simulation flow. A simulation loop (step 1) where all the necessary functions are performed is defined by the length of a power control step. In WCDMA the time between two power control steps is 0.625 ms whereas in GSM it is 480 ms /22/. Therefore, simulating one second in real time in a WCDMA system simulator requires the execution of the simulation loop 1600 times, when the corresponding amount in the GSM is 2. Overall, the simulation process of the WCDMA system simulator is much more complex than the corresponding GSM simulator.

In step 2, each base station is processed. Base station algorithms for connections are performed and radio receiver algorithms are called for the base station itself. The radio receiver algorithms consist of: maximal ratio combining, multi-user detection, and antenna diversity. In terminal calculation (step 3) all active terminals are processed. The function handles power control, handover, and mobility routines. In step 4 the base station (radio network) controllers execute the necessary algorithms for each connection, in order to check for the need for handovers. If a handover is needed, it is performed by this function. Connection calculation (step 5) performs actions for all connections; if there are connections in terminals that are not active, they are deleted. In interference calculation (step 6) all interferences caused by base stations and terminals are calculated. Traffic generation (step 7) creates new terminals and connections according to the probability given as a parameter by the user.

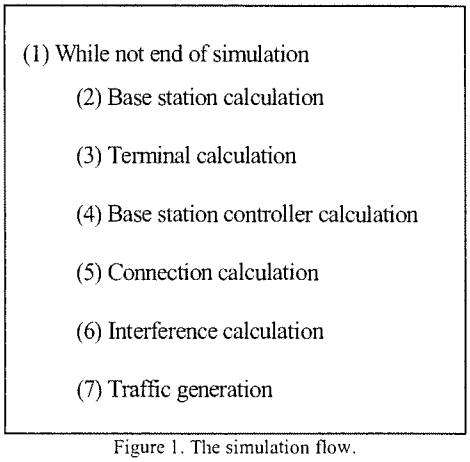


Figure 1. The simulation flow.

Test runs showed that the most time-consuming parts of the simulator are terminal calculation (step 3) and interference calculation (step 6) /9/. These two parts consumed over 75% of the total execution time: terminal calculation and interference calculation consumed 20% and 55% respectively. Therefore, in the parallelization process the main focus was on these two parts. In the following chapter, terminal calculation and interference calculation are explained with greater detail, and the parallel approaches taken in both of the cases are shown.

The parallel implementation was done with POSIX threads. Since the thread library is based on the POSIX standard, the portability of the software is good /5/. On the other hand, the other and more powerful thread library, Solaris threads, can be used in Sun Microsystem's workstations and servers only /15/. The use of threads is based on a thread-process concept /9/. The threads operate inside a single process sharing all the resources allocated to the process. This enables a more efficient handling of context switching and interprocess communication /13/. Since all threads are inside a process, the execution of threads does not require context switches since they are usually done between processes. Threads are divided into processors according to the scheduling policy of the operating system.

Since all threads share the resources allocated for the process, a synchronization method has to be implemented to prevent the concurrent accessing of these resources /18/. For example, the same memory location cannot be accessed by two or more threads at the same time. Otherwise, the content of the memory location becomes undeterminable. Even though threads generate overhead in the form of synchronization, they provide an effective way to implement parallelism into an application. The other possibility is the use of the PVM or the MPI libraries in a distributed memory machine /4, 14/. In a distributed memory machine, message passing generates a great deal of overhead. However, due to the architecture of the processor network, distributed memory machines can have significantly more processors than shared memory machines /7/. This fact makes the distributed memory machine a suitable platform for calculation intensive problems, since they require large number of processors in order to compute within a reasonable time frame.

In the simulator presented in this paper, threads were used with a shared memory machine to implement the parallel version of the simulator. A shared memory machine was chosen over the distributed memory machine because a lot of data needed to be transferred before, during and after the parallel calculation. In a distributed memory machine, this would have generated a great deal of overhead causing the parallel version to perform poorly /21/.

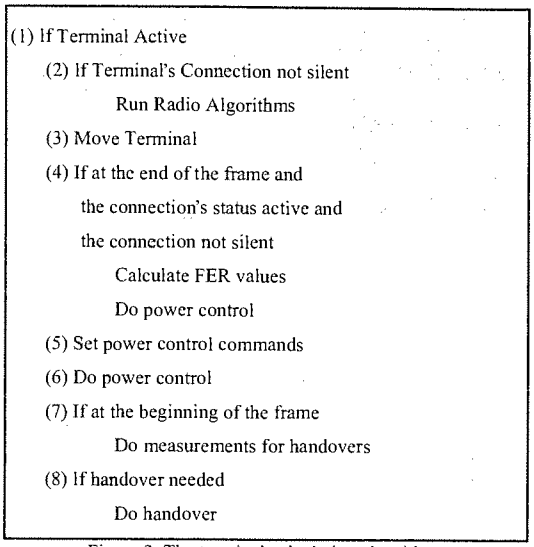


Figure 2. The terminal calculation algorithm.

Terminal calculation

The purpose of terminal calculation is to execute all the necessary functions for each terminal in the network. Figure 2 represents the actions taken in terminal calculation. First, only the active terminals are processed (step 1). A terminal needs to have a physical connection in downlink or uplink to be considered active. If the connection is not silent (step 2), i.e. data is being transmitted over the links, radio algorithm for the connection is performed. In step 3, the terminal is moved according to predefined move procedure depending on the network structure (micro or macrocell). If all the conditions in step 4 are true, Frame Error Ratio calculations and power control routines are performed. In step 5 power control commands for the downlink are set. The commands are set based on the comparison to Signal-to-Interference Ratio (SIR) and the threshold. The power control commands are set in step 5 and executed in step 6. The need for handover is determined in step 7 where all the necessary measurements are carried out. If the handover is required it is executed in step 8.

For terminal calculation a special work balancing algorithm was implemented based on the structure of the code. The parallel terminal calculation algorithm is shown in Figure3. In step 1, each thread waits until it receives a signal from the main program as a notification to start its execution. After that, in step 2, each thread calculates the amount of terminals it will process in phase one. The number of terminals processed in phase one is based on a simple division M/n, where M is the mean load of terminal and n the number of threads. The mean load of terminal indicates the average amount of active terminals in the network, i.e. the mean number of terminals that need to be processed. If there is a remainder r, r threads will have an extra terminal to process. Terminals are stored in a static array. At the beginning of the simulation, all the t terminals are stored in t first positions of the array. However, when the simulation goes on the locations of the terminals can be outside the first t positions. Nevertheless, the simulator attempts to use all available location inside the t positions. Based on this fact as well as knowledge concerning the average number of terminals in the array, the division is made. Each thread is assigned part of the reduced array (array of locations 1 to *mean load*). Since all the terminals are not inside this range, the thread that first completes its work will process the remaining part of the array, *mean load+1* to *end of the array*. It has been noted that the thread processing the outside part of the array usually finishes its work before the other threads. This can be explained by the simple fact that outside the mean load there are not many terminals to process, and processing a single terminal is a much more time-consuming event than going through an empty array of terminals. The actual processing takes place in step 3. For each terminal, the sequential terminal calculation algorithm is performed. When a thread has completed all its work it will suspend itself and waits for the signal from the main program in step 5. The last thread completing its work will send a signal to the main program indicating that all threads have finished their work before suspending its execution in step 4.

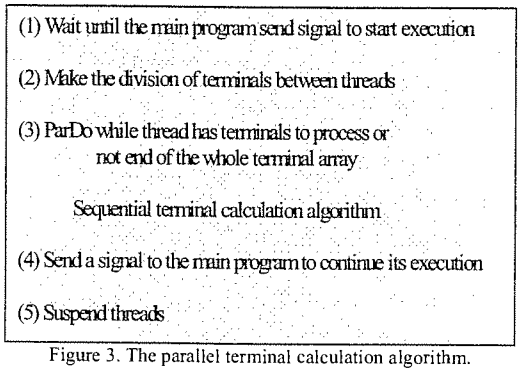


Figure 3. The parallel terminal calculation algorithm.

Interference calculation

The structure of interference calculation is represented in Figure4. The interference calculation algorithm is used to calculate all the interfereces caused by either mobile or base station. As it has been noted previously, the power control, of which interference calculation is a part, is an essential part of WCDMA network. Execution of this algorithm takes about 55% of the total execution time. The algorithm consists of the following parts: In steps 1 and 2, all interference variables are set to zero. The outer loop in step 3 goes through all base stations. For each base station all mobile stations are processed in step 4. In step 5, uplink interference caused by the base station to the mobile station is calculated, whereas in step 6, downlink interference caused by a single mobile station to a base station is determined. Finally, in step 7 and 8, uplink and downlink interferences are added together for each base station and terminal.

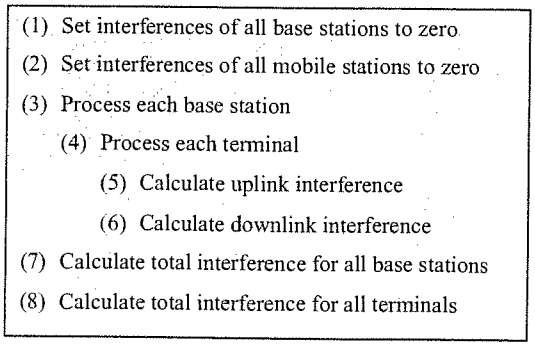


Figure 4. The interference calculation algorithm.

For the parallel approach to interference calculation a modified WorkPool algorithm was applied /10, 11, 12/. In the WorkPool algorithm the work is divided according to the actual work and not with the simple equation: *work / number of processors*. Figure5 illustrates the parallel algorithm for interference calculation. In the algorithm in step 1 all threads wait until a signal is sent by the main program. Then the actual parallel calculation takes place. In this case the workload is divided according to the base stations. Each thread processes one base station at a time; all base stations are in a pool where threads can access one at a time to process. Threads keep fetching new base stations until the pool is empty. With the use of a pool, where all the base stations are located, better performance is achieved. Until the pool is empty, the workload is divided between threads as equally as possible. However, the use of a WorkPool algorithm introduces a slight amount of overhead to the actual calculation. Since threads fetch a new base station from the pool, a method has to be implemented to prevent the processing of the same base station by two or more threads. To fulfill this requirement a synchronization method is needed. For interference calculation the method used was a mutual exclusion lock (mutex lock). With the mutex lock a part of the code can be protected so that only one thread can execute the protected portion of the code at a time. While a thread is executing the protected part of the code, other threads wanting to enter the same area will be blocked until the mutex lock is released. In step 3, a signal is sent to the main program, which then continues its execution. Finally, the last thread, which sent the signal, also suspends its execution, just like all the other threads which had already reach step 4 earlier.

```
(1) Wait until the main program sends a signal to start execution
(2) ParDo until all base stations have been processed
        Do loop until all mobile stations have been processed
            Calculate uplink interference
            Calculate downlink interference
(3) Send a signal to the main program to continue its execution
(4) Suspend threads
```

Figure 5. The parallel interference calculation algorithm.

RESULTS

The test runs were performed on an SMP machine with 6 processors. The processors were Sun Microsystem's 333 MHz Ultra Sparc chips. All the processors shared a memory of 2 gigabytes. The workstation was not dedicated to the parallel simulation of a WCDMA simulator. Therefore, the results obtained have some overhead in them because of the other users' programs, which were run simultaneously with the parallel simulator.

As a test environment, a network of 9 base stations was created. For each base station two sectors were constructed. Therefore, the total number of base station entities to simulate totaled 18. The maximum number of terminals in the network was set to 650. However, the average number of active terminals in the network at a time was about 100. The simulation time was 30 seconds in real time. A simulation of only half a minute is not enough to provide accurate results for research purposes but it was enough to give insight into the performance of parallel processing. In general, it can be said that if the simulator performs well with a short simulation time it will perform at least as good or even better with a longer simulation time. This is because the sequential part does not have such a significant effect on long simulation times as it does on short simulation times. The WCDMA simulator will produce accurate results with the simulation time of 120 seconds and up.

Table 1 shows all the execution times of terminal calculation, interference calculation and the total execution time of the whole simulator, with different numbers of processors. Figure 6 represents the calculated speedups based on the execution times of Table 1. The speedups are calculated against the sequential execution time achieved with a simulator which did not include any parallel code.

Table 1. The execution times (in seconds) of the simulator.

# of Threads	Terminal calculation	Interference calculation	Total Execution time
Seq	1030	2450	4780
2	600	1280	3100
3	460	920	2630
4	410	820	2460
5	470	780	2490
6	470	740	2440

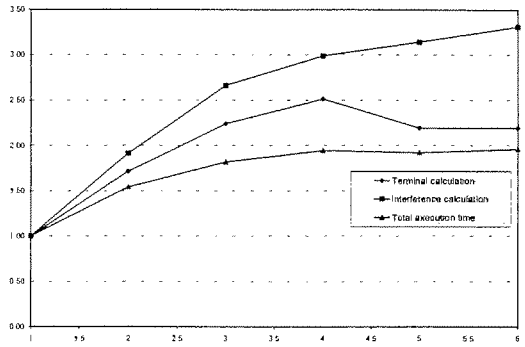


Figure 6. Achieved speedups.

Terminal calculation

The best speedup of 2.5 was achieved with 4 threads. The corresponding execution times were 410 seconds for the parallel implementation and 1030 seconds for the sequential simulator. By using more than 4 processors the speedup started to decline, the reason being the two mutex locks in the parallel terminal calculation algorithm. Even though the work balancing did not require any synchronization, the actual parallel terminal calculation algorithm did. Two parts of the code were protected with the locks to prevent the concurrent memory access to a shared memory block. The other cause of the parallel terminal calculation algorithm's poor performance was the fact that the execution of the algorithm does not take long even without any parallel options. Therefore, since the amount of work is relatively small, the parallelization does not have such a strong effect as in interference calculation.

Interference calculation

The speedups achieved in interference calculation were better than in terminal calculation. As can be seen from Figure 6, the speedup increases when processors are applied all the way to 6. The amount of work and the structure of the interference calculation algorithms are the reasons for this. There is enough work for all six processors, whereas in terminal calculation, the use of six processors was not productive. On the other hand, the algorithm for interference calculation was more optimal for parallelization than the terminal calculation algorithm.

The best speedup of 2.4 was achieved with 6 processors. However, the efficiency of the parallel execution is not high. With the use of six processors, the speedup of 4 or more would be considered good. In this case the simulator was implemented first without any consideration to the parallel execution. Therefore, in many instances, better performance figures would have been achieved if the structure of the code had been more

optimal for parallelization. Since the parallel implementation was done by modifying the existing sequential code, compromises had to be made at the cost of speed. Especially, in the case of interference calculation algorithm, where a number of mutex locks had to be implemented to guarantee the correct execution of the code and the memory coherence.

Total execution time

In the sequential simulator, the two parts which were parallelized consumed about 75% of the total execution time. With the use of multiple processors their share of the total execution time were dropped below 50%. Therefore, the problem that remains in the parallelized simulator is the sequential part. Time spent in executing the sequential part is over 50% of the total execution time. The two parallelized parts are the only ones that can be made to run in parallel. The remaining sequential part consists of routines and functions that either cannot be parallelized or the parallel implementation would cause significant overhead and increase the execution time rather that decrease it. In conclusion, the sequential part has to be modified so that a parallel approach can be taken and the execution time can be decreased, otherwise the speedups achieved cannot be improved. With the use of four or more processors the speedup remains at the same level. This is the level where the sequential part starts to dominate the execution time. However, the best speedup of 2.0 is achieved with the maximum number of processors (6).

Amdahl's law can be used to estimate the maximum speedup of a parallel application based on the execution times of the sequential and parallel parts of the application /2, 5/. Applying Amdahl's law to the parallel simulator presented in this paper gives the following results:

$S_{max} \approx 4780s / (1230 + 3550 / 6) \approx 2.6$

Amdahl's law indicates that the maximum speedup that can be achieved with the simulator is 2.6. The maximum speedup which was achieved was 2.0. The difference of 0.6 can be explained by the mutex locks and other overhead generated by the parallel execution. Therefore, the speedup of 2 with 6 processors is not as poor as it seems initially.

CONCLUSIONS

In this paper a parallelization process for a WCDMA system simulator was presented. The need for a parallel implementation of the simulator was great due to the excessive execution times of the sequential version. POSIX threads were selected as a method to physically implement parallelism into the simulator. The use of POSIX threads is practical due to their portability to nearly all UNIX platforms and operating systems. The shared memory environment was the optimal foundation for the use of threads because of their nature of operation.

The two most time-consuming parts of the WCDMA system simulator were parallelized: terminal calculation and interference calculation. The terminal calculation algorithm performs operations to all active terminals (mobile stations): it moves terminals, makes handovers, sets and executes power control commands. The interference calculation algorithm, on the other hand, calculates interferences caused by the base stations and terminals. For terminal calculation, a static work balancing algorithm was developed to handle the equal division of workload. The workload is divided between threads according to the mean load of terminals. In addition, the work balancing algorithm used knowledge about the structure of the array where terminals are stored to increase performance. Contrary to terminal calculation, interference calculation has a dynamic work balancing algorithm; base stations are dynamically divided between threads one at a time. Due to the amount of work in each base station, a dynamic work balancing algorithm proved to perform better than a static work balancing algorithm, similar to the one in terminal calculation.

The results were very promising. However, the major disadvantage in the simulator was the structure into which it was coded. Due to this fact, the parallelization of the whole simulator was unattainable since the structure of the code was optimized for sequential execution. Nevertheless, the results showed that execution times can be decreased with the use of parallel processing in an existing sequential simulator. The speedup of 2.0 for the whole simulator was achieved with 6 threads. Better speedups would have been achieved if the sequential part had not been so large; over 50 % of the total execution time was spent in executing the sequential proportion of the code. The best speedups of terminal calculation and interference calculation were 2.5 and 3.3 respectively. In conclusion, it can be said that with the correct use of work balancing algorithms and a parallel computer, the execution times of the WCDMA system simulator can be decreased considerably.

REFERENCES

/1/ Adachi, F., Sawahashi, M., et. al.: Wideband DS-CDMA for Next-Generation Mobile Communications Systems. IEEE Communications Magazine, September, 1998. pp. 56-69.

/2/ Akl, S.G. Parallel Computation: Models And Methods. USA, Prestice Hall PTR, 1997. ISBN 0-13-147034-5.

/3/ Berendt, A., McClure, B.: Third Generation. Telecommunications, September, 1998. pp. 28-34.

/4/ Bruck, J., Dolev, D., et. al.: Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. Journal of Parallel and Distributed Computing, 1997. Vol. 40, pp. 19-34.

/5/ Butenhof, D.R. Programming with POSIX Threads. USA, Addison-Wesley, 1997. ISBN 0-201-63392-2.

/6/ Dahlman, E., Gudmundson, B., et.al: UMTS/IMT-2000 Based on Wideband CDMA. IEEE Communications Magazine, September, 1998. pp. 70-80.

/7/ El-Rewini, H., Lewis, T.G. Distributed and Parallel Computing. USA, Manning Publications, 1997. ISBN 0-13-795592-8.

/8/ Hendessi, F., Sheikh, A.U.H., et al.: A TDMA-CDMA Cellular System. Proceedings of Vehicular Technology Conference (VTC'97), 1997. Vol. 1, pp. 373-376.

/9/ Huttunen, P: Improving the Performanee of a WCDMA System Simulator Through Parallel Computing Techniques. Master's Thesis, Lappeenranta University of Technology, Finland, March, 1999.

/10/ Huttunen, P., Ikonen, J., et. al.: Using Cray T3E for parallel calculation of cellular radio coverage. Proceedings of European Simulation Congress (Eurosim'98), 1998. Vol. 1, pp. 27-32.

/11/ Huttunen, P., Ikonen, J., et. al.: Parallelization of propagation model simulation. Proceedings of European Simulation Symposium (ESS'98), 1998. pp. 321-325.

/12/ Huttunen, P., Porras, J., et. al.: Simulation of mobile networks in multiprocessor environments. CSC News, February, 1999. pp. 12-13.

/13/ Kleiman, S., Shah, D., et. al. Programming with Threads. USA, Prentice Hall, 1996. ISBN 0-13-172389-8.

/14/ Lauria, M., Chien, A.: MPI-FM: High Performance MPI on Workstation Clusters. Journal of Parallel and Distributed Computing, 1997. Vol. 40, pp 4-18.

/15/ Lewis, B., Berg D.J. Threads Primer. USA: SunSoft Press, 1996. ISBN 0-13-443698-9.

/16/ Milstein, L.B., Simon M.K.: Spread Spectrum Communications. The Mobile Communication Handbook. USA, CRC Press, 1996. pp. 152-165. ISBN 0-8394-8573-3.

/17/ Miya, K., et al.: CDMA/TDD Cellular Systems for the 3rd Generation Mobile Communication. Proceedings of Vehicular Technology Conference (VTC'97), 1997. Vol. 2, pp. 820-824.

/18/ Norton, S.J., Dipasquale, M.D. Thread Time. USA, Prestice Hall, 1997. ISBN 0-13-190067-6.

/19/ Ojanperä, T., Prasad, R. (editors). Wideband CDMA for Third Generation Mobile Communications. USA, Artech House, 1998. ISBN 0-89006-735-X.

/20/ Ross, A.H.M., Gilhousen K.S.: CDMA Technology and the IS-95 North American Standard. The Mobile Communication Handbook. USA, CRC Press, 1996. pp. 430-448. ISBN 0-8394-8573-3.

/21/ Sohn, A., Sato, M., et. al.: Data and Workload Distribution in a Multithreaded Architecture. Journal of parallel and distributed computing, 1997. Vol. 40, pp. 256-264.

/22/ Viterbi, A.J. CDMA: principles of spread spectrum communication. USA, Addison-Wesley, 1995. ISBN 0-201-63374-4.

/23/ Westman, T., Holma H.: CDMA System for UMTS High Bit Rate Services. Proceedings of Vehicular Technology Conference (VTC'97), 1997. Vol. 2, pp. 825-829.

87. ESKELINEN, HARRI. Tuning the design procedures for laser processed microwave mechanics. 1999. 172 s. Diss.

88. ROUVINEN, ASKO. Use of neural networks in robot positioning of large flexible redundant manipulators. 1999. 71 s. Diss.

89. MAKKONEN, PASI. Artificially intelligent and adaptive methods for prediction and analysis of superheater fireside corrosion in fluidized bed boilers. 1999. 187 s. Diss.

90. KORTELAINEN, JARI. A topological approach to fuzzy sets. 1999. U.s. Diss.

91. SUNDQVIST, SATU. Reaction kinetics and viscosity modelling in the fusion syntheses of Ca- and Ca/Mg-resinates. 1999. U.s. Diss.

92. SALO, JUSSI. Design and analysis of a transversal-flux switched-reluctance-linear-machine pole-pair. 1999. 156 s. Diss.

93. NERG, JANNE. Numerical modelling and design of static induction heating coils. 2000. 86 s. Diss.

94. VARTIAINEN, MIKA. Welding time models for cost calculations in the early stages of the design process. 2000. 89 s., liitt. Diss.

95. JERNSTRÖM, EEVA. Assessing the technical competitiveness of printing papers. 2000. 159 s., liitt. Diss.

96. VESTERINEN, PETRI. On effort estimation in software projects. 2000. U.s. Diss.

97. LUUKKO, JULIUS. Direct torque control of permanent magnet synchronous machines – analysis and implementation. 2000. 172 s. Diss.

98. JOKINEN, ARTO. Lobbying as a part of business management. 2000. 244 s. Diss.

99. JÄÄSKELÄINEN, EDUARD. The role of surfactant properties of extractants in hydrometallurgical liquid-liquid extraction processes. 2000. U.s. Diss.

100. Proceedings of 3rd Finnish-French Colloquium on Nuclear Power Plant Safety. 2000. 118 s.

101. TANSKANEN, PASI. The evolutionary structural optimization (ESO) method: theoretical aspects and the modified evolutionary structural optimization (MESO) method. 2000. 67 s., liitt. Diss.

102. JERNSTRÖM, PETTERI. The effects of real-time control of welding parameters on weld quality in plasma arc keyhole welding. 2000. 69 s., liitt. Diss.

103. KAARNA, ARTO. Multispectral image compression using the wavelet transform. 2000. U.s. Diss.

104. KOTONEN, ULLA. Rahavirta-analyysit, erityisesti kassavirtalaskelma, kunnan talouden ohjauksen apuvälineenä. 2000. 209 s., liitt. Väitösk.

105. VARIS, JUHA. A novel procedure for establishing clinching parameters for high strength steel sheet. 2000. 84 s., liitt. Diss.

106. PÄTÄRI, EERO. Essays on portfolio performance measurement. 2000. 201 s. Diss.

107. SANDSTRÖM, JAANA. Cost information in engineering design – potentials and limitations of activity-based costing. 2001. 143 s., liitt. Diss.

108. TOIVANEN, JOUKO. Balanced Scorecardin implementointi ja käytön nykytila Suomessa. 2001. 216 s. Väitösk.

109. PESONEN, MAUNO. Applying AHP and A'WOT to strategic planning and decision making: case studies in forestry and forest industry. 2001. U.s. Diss.

110. Proceedings of Fifth International Seminar on Horizontal Steam Generators. Ed. by Juhani Vihavainen. 2001. 255 s.

111. LAINE, PERTTI. Kohti vesiensuojelun aikaa: veden laadun muutokset eteläisellä Saimaalla. 2001. 264 s. Väitösk.

112. SILVENTOINEN, PERTTI. Electromagnetic compatibility and EMC-measurements in DC-voltage link converters. 2001. 115 s. Diss.

113. TERVONEN, ANTERO. Laadun kehittäminen suomalaisissa yrityksissä. 2001. 206 s. Väitösk.

114. SALMINEN, ANTTI. The effects of filler wire feed on the efficiency, parameters and tolerances of laser welding. 2001. 82 s., liitt. Diss.

115. HORTTANAINEN, MIKA. Propagation of the ignition front against airflow in packed beds of wood particles. 2001. U.s. Diss.

116. IKONEN, JOUNI. Improving distributed simulation in a workstation environment. 2001. U.s. Diss.

117. WU, HUAPENG. Analysis, design and control of a hydraulically driven parallel robot manipulator. 2001. U.s. Diss.

118. REUNANEN, ARTTU. Experimental and numerical analysis of different volutes in a centrifugal compressor. 2001. 150 s. Diss.

119. TAAVITSAINEN, VELI-MATTI. Strategies for combining soft and hard modelling in some physicochemical problems. 2001. U.s. Diss.

120. SAVOLAINEN, RAIJA. The use of branched ketene dimers in solving the deposit problems related to the internal sizing of uncoated fine paper. 2001. U.s. Diss.

121. SARAVIRTA, ALI. Project success through effective decisions: case studies on project goal setting, success evaluation and managerial decision making. 2001. 286 s. Diss.

122. BLOMQVIST, KIRSIMARJA. Partnering in the dynamic environment: the role of trust in asymmetric technology partnership formation. 2002. 296 s., liitt. Diss.

123. KARVONEN, VESA. Development of fiber recovery process. 2002. U.s. Diss.

124. KÄYHKÖ, JARI. The influence of process conditions on the deresination efficiency in mechanical pulp washing. 2002. 87 s., liitt. Diss.

125. SAVOLAINEN, PEKKA. Modeling of non-isothermal vapor membrane separation with thermodynamic models and generalized mass transfer equations. 2002. 179 s. Diss.

126. KÄRKKÄINEN, HANNU. Customer need assessment: Challenges and tools for product innovation in business-to-business organizations. 2002. U. s. Diss.

127. HÄMÄLÄINEN, MARKKU. Spray coating technique as a surface treatment for woodcontaining paper grades. 2002. 121 s. Diss.

129. KUOSA, MAUNU. Numerical and experimental modelling of gas flow and heat transfer in the air gap of an electric machine. 2002. 97 s. Diss.

131. SUNDQVIST, SANNA. Market orientation in the international context: Antecedents, consequences and applicability. 2002. U. s. Diss.