Diploma Thesis

Applied High Performance Computing Using R

Stefan Theußl

27.09.2007

Supervisor: univ. Prof. Dipl. Ing. Dr. Kurt Hornik

Department of Statistics and Mathematics
Vienna University of Economics and Business Administration
Augasse 2-6
A-1090 Vienna

Abstract

Applied High Performance Computing Using R

In the 1990s the Beowulf project smoothed to way for massively parallel computing as access to parallel computing power became affordable for research institutions and the industry. But the massive breakthrough of parallel computing has still not occurred. This is because two things were missing: low cost parallel computers and simple to use parallel programming models. However, with the introduction of multicore processors for mainstream computers and implicit parallel programming models like OpenMP a fundamental change of the way developers design and build their software applications is taken place—a change towards parallel computing.

This thesis gives an overview of the field of high performance computing with a special focus on parallel computing in connection with the R environment for statistical computing and graphics. Furthermore, an introduction to parallel computing using various extensions to R is given.

The major contribution of this thesis is the package called **paRc**, which contains an interface to OpenMP and provides a benchmark environment to compare various parallel programming models like MPI or PVM with each other as well as with highly optimized (BLAS) libraries. The dot product matrix multiplication was chosen as the benchmark task as it is a prime example in parallel computing.

Eventually a case study in computational finance is presented in this thesis. It deals with the pricing of derivatives (European call options) using parallel Monte Carlo simulation.

Kurzfassung

Angewandtes Hochleistungsrechnen unter Verwendung von R

In den 1990er Jahren ebnete das Beowulf Projekt den Weg für massives Parallelrechnen, weil dadurch der Zugang zu parallelen Rechenresourcen für viele Forschungsinstitute und der Industrie erst leistbar wurde. Trotzdem war der große Durchbruch lange nicht erkennbar. Ausschlaggebend dafür waren zwei Aspekte: Erstens gab es bisher keine günstigen Parallelrechner und zweitens waren auch keine einfachen Parallelprogrammiermodelle

vorhanden. Die Einführung von Multicoreprozessoren für Desktopsysteme und impliziter Programmiermodelle wie OpenMP führte zu einem Umbruch, der auch Auswirkungen auf die Softwareentwicklung hat. Der Trend geht dabei immer mehr in Richtung Parallelrechnen.

Diese Diplomarbeitet bietet einen Überblick über das Feld des Hochleistungsrechnens und im Speziellen des Parallelrechnens unter Verwendung von R, einer Softwareumgebung für statistisches Rechnen und Grafiken. Darüberhinaus beinhaltet diese Arbeit eine Einführung in paralleles Rechnen unter Verwendung mehrerer Erweiterung zu R.

Der Kern dieser Arbeit ist die Entwicklung eines Paketes namens **paRc**. Diese Erweiterung beinhaltet eine Schnittstelle zu OpenMP und stellt eine Benchmark Umgebung zur Verfügung, mit der verschiedene Paralleleprogrammiermodelle wie MPI oder PVM sowie hochoptimierte Bibliotheken (BLAS) miteinander verglichen werden können. Als Aufgabe für diese Benchmarks wurde das klassische Beispiel aus dem Bereich des Parallelrechnens, nämlich die Matrix Multiplikation, gewählt.

Schlussendlich wird in dieser Arbeit eine Fallstudie aus der computationalen Finanzwirtschaft präsentiert. Im Mittelpunkt dieser Fallstudie steht die Bewertung von Derivaten (Europäische Call Optionen) unter Verwendung paralleler Monte Carlo Simulation.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 2
- 1.3 Organization of this Thesis . . . 2

2 Parallel Computing **3**
- 2.1 Introduction . . . 3
- 2.2 Terms and Definitions . . . 5
  - 2.2.1 Process vs. Thread . . . 5
  - 2.2.2 Parallelism vs. Concurrency . . . 5
  - 2.2.3 Computational Overhead . . . 6
  - 2.2.4 Scalability . . . 7
- 2.3 Computer Architecture . . . 7
  - 2.3.1 Beyond Flynn's Taxonomy . . . 7
  - 2.3.2 Processor and Memory . . . 9
  - 2.3.3 High Performance Computing Servers . . . 11
- 2.4 MIMD Computers . . . 13
  - 2.4.1 Shared Memory Systems . . . 14
  - 2.4.2 Distributed Memory Systems . . . 14
- 2.5 Parallel Programming Models . . . 16
  - 2.5.1 Fundamentals of Message Passing . . . 17
  - 2.5.2 The Message Passing Interface (MPI) . . . 18
  - 2.5.3 Parallel Virtual Machine (PVM) . . . 20
  - 2.5.4 OpenMP . . . 21
- 2.6 Design of Parallel Programs . . . 23
  - 2.6.1 Granularity of Parallel Programs . . . 23
  - 2.6.2 Organization of Parallel Tasks . . . 24
  - 2.6.3 Types of Achieving Parallelism . . . 25
- 2.7 Performance Analysis . . . 26
  - 2.7.1 Execution Time and Speedup . . . 26
  - 2.7.2 Amdahl's Law . . . 26

2.8 Hardware and Software Used . . . 27
2.8.1 High Performance Computing Servers . . . 27
2.8.2 Utilized Software . . . 29
2.9 Conclusion . . . 30

3 High Performance Computing and R 31
3.1 Introduction . . . 31
3.2 The R Environment . . . 31
3.3 The Rmpi Package . . . 32
3.3.1 Initialization and Status Queries . . . 33
3.3.2 Process Spawning and Communication . . . 34
3.3.3 Built-in High Level Functions . . . 36
3.3.4 Other Important Functions . . . 37
3.3.5 Conclusion . . . 38
3.4 The rpvm Package . . . 38
3.4.1 Initialization and Status Queries . . . 39
3.4.2 Process Spawning and Communication . . . 40
3.4.3 Built-in High Level Functions . . . 40
3.4.4 Conclusion . . . 44
3.5 The snow Package . . . 45
3.5.1 Initialization . . . 46
3.5.2 Built-in High Level Functions . . . 46
3.5.3 Fault Tolerance . . . 48
3.5.4 Conclusion . . . 48
3.6 paRc—PARallel Computations in R . . . 49
3.6.1 OpenMP Interface Functions . . . 49
3.6.2 High Level OpenMP Functions . . . 50
3.6.3 Benchmark Environment . . . 51
3.6.4 Environment for Option Pricing . . . 54
3.6.5 Other Functions . . . 57
3.7 Other Packages Providing HPC Functionality . . . 58
3.8 Conclusion . . . 59

4 Matrix Multiplication 60
4.1 Introduction . . . 60
4.2 Notation . . . 60
4.2.1 Column and Row Partitioning . . . 61
4.2.2 Block Notation . . . 61
4.3 Basic Matrix Multiplication Algorithm . . . 62
4.3.1 Dot Product Matrix Multiplication . . . 62
4.3.2 C Implementation . . . 63

4.3.3 Using Basic Linear Algebra Subprograms 63
4.3.4 A Comparison of Basic Algorithms with BLAS 64
4.4 Parallel Matrix Multiplication 65
4.4.1 Benchmarking Matrix Multiplication 66
4.4.2 OpenMP 67
4.4.3 MPI 69
4.4.4 PVM 73
4.5 Comparison of Parallel Routines 75
4.5.1 Shared Memory Platform 76
4.5.2 Distributed Memory Platform 77
4.6 Conclusion 80

5 Parallel Monte Carlo Simulation 87
5.1 Introduction 87
5.2 Option Pricing Theory 88
5.2.1 Derivatives 88
5.2.2 Black-Scholes Model 89
5.2.3 Monte Carlo Evaluation of Option Prices 94
5.3 Monte Carlo Simulation 96
5.3.1 Implementation in R 97
5.3.2 Parallel Monte Carlo Simulation 99
5.3.3 Notes on Parallel Pseudo Random Number Generation 101
5.3.4 Results of Parallel Monte Carlo Simulation 101
5.4 Conclusion 103

6 Conclusion and Future Work 105
6.1 Summary 105
6.2 Economical Coherences 106
6.3 Outlook 107
6.4 Acknowledgments 107

A Installation of Message Passing Environments 109
A.1 LAM/MPI 109
A.2 PVM 110

B Sun Grid Engine 111
B.1 LAM/MPI Parallel Environment 111
B.2 PVM Parallel Environment 112

C Benchmark Definitions **114**
C.1 Matrix Multiplication 114
C.2 Option Pricing 116

Chapter 1

Introduction

1.1 Motivation

During the last decades the demand for computing power has steadily increased, since problems like climate prediction, data mining, difficult computations in physics, or optimizations in various fields make use of more accurate and therefore time consuming models. It is obvious that minimizing time needed for these calculations has become an important task. Therefore a new field in computer science has come up—high performance computing.

Being able to write code which can be executed in parallel is a key ability in this field because high performance computing servers provide in general more than just one processor. But one has to consider that parallelization of sequential code is a difficult task and is indeed another access in writing programs. To help scientists and software developers standards and parallelizing techniques like MPI or PVM have been introduced to make their work easier. Nevertheless, sequential programs have to be rewritten, or completely new applications have to be designed to make use of this enormous parallel computing power. Another approach has become popular again recently—compiler driven parallelization. Though writing parallel programs can be achieved more easily the performance cannot be compared to code which has been parallelized manually. All in all, writing efficient code remains a challenging, time consuming task and needs years of experience.

Therefore, the major aim of this thesis is not only to provide a good overview of the work which has been done in this field but more importantly to become familiar with the state of the art of parallel computing.

Moreover, a lot of time consuming models exist in a scientific field called computational statistics; a field which has many connections to other scientific areas (e.g., economics, physics, biology or engineering). As a com-

putational environment a software and language called R (R Development Core Team (2007b)) is available. This project is on its way to provide high performance computing to statistical computing through various extensions. To become familiar with the capabilities of R in this area is of major interest as it would be a key for solving many problems not only in statistics but also in all other areas where computational statistics plays a major role. In this thesis parallel Monte Carlo simulation used in finance to price options has been chosen as a case study.

1.2 Contributions

In the course of this thesis an extension to the R environment for statistical computing has been developed—the **paRc** package (Theussl (2007b)). First, it provides a benchmark environment for identifying performance gains of parallel algorithms. Results of parallel matrix multiplication were produced using this benchmark environment. Second, a first attempt to use OpenMP (OpenMP Architecture Review Board (2005)) in combination with R has been made and a C interface to OpenMP library calls has been implemented. Finally, a framework for pricing options with parallel Monte Carlo simulation has been developed.

1.3 Organization of this Thesis

This thesis is organized as follows. First we briefly summarize the fundamentals of high performance computing and parallel processing in Chapter 2. In the course of this chapter computer architectures and programming models are described as well as the principles of performance analysis.

Chapter 3 includes explanations of how to use these paradigms in R. This part of this thesis provides an overview of the available extensions supplying parallel computing functionality to R.

In the remaining chapters case studies for high performance computing are presented. In Chapter 4 benchmarks of the parallel matrix multiplication are compared for each parallel programming model as well as for highly optimized libraries. The aim is to point out which programming models available in R are most promising for further work. The case study presented in Chapter 5 deals with the pricing of derivatives (in particular of European call options) using parallel Monte Carlo simulation.

Eventually Chapter 6 summarizes the findings and give economical interpretations as well as an outlook to future work.

Chapter 2

Parallel Computing

2.1 Introduction

In 1965 a director in the Fairchild Semiconductor division of Fairchild Camera and Instrument Corporation predicted: "The complexity [of integrated circuits] for minimum costs has increased at a rate of roughly two per year [...] over the short term this rate can be expected to continue, if not increase."(Moore (1965)) This soon became known as Moore's Law. And indeed for nearly forty years the advance predicted by Moore has taken place (the overall growth of the world's most powerful computers has approximately doubled every 18 months).

Furthermore, with greater speed more memory is needed, otherwise one has the situation that a computer capable of performing trillions of operations in every second only has access to a small memory which is rather useless for performing huge data driven calculations. In addition to that the amount of data being available to analysts today has rapidly increased. Data collection and mining is done nearly everywhere. This bulk of data implies that enough memory is available to process it. With this development a new field in computer science established and has become increasingly important over the last decades—high performance computing.

High Performance Computing

The term high performance computing (HPC) refers to the use of (parallel) supercomputers and computer clusters (Wikipedia (2007a)). Indeed it can be explained as computing on a high performance computing machine. This machines can be clusters of workstations or huge shared memory machines (see Section 2.3.3 for more details on high performance computing servers).

Furthermore, HPC is a branch of computer science that concentrates

on developing high performance computers and software to run on these computers. An important area of this discipline is the development of parallel processing algorithms or software referred to as **parallel computing**.

Trends Constituting High Performance Computing

Now, more than 40 years after Moore's testimony desktop systems with two to four processors and lots of memory are available. This means that standard stock computers can be compared with high performance workstations of just a few years ago.

Factors which constitute this trend are (Hennessy and Patterson (2007)):

- a slowdown in uniprocessor performance arising from diminishing returns in exploiting instruction level parallelism,
- a growth in servers and server performance,
- a growth in data intensive applications,
- the insight that increasing performance on the desktop is less important,
- an improved understanding of how to use multiprocessors effectively, especially in server environments where there is significant thread level parallelism,
- the advantages of leveraging a design investment by replication rather than unique design—all multiprocessor designs provide such leverage.

With the need to solve large problems and the availability of adequate workstations, large-scale parallel computing has become more and more important. Some examples to illustrate this need in science and industry include:

- data mining for large data sets,
- algorithms for solving $\mathcal{NP}$-complete problems,
- climate prediction,
- computations in biology, chemistry and physics,
- cryptography,
- and astronomy.

The R project for Statistical Computing and Graphics is an open source software environment available for different platforms. With the development mentioned above there have come up some extensions (called packages) for high performance computing. That means R is already prepared to be used in this field. A more detailed overview of high performance computing in connection with R can be found in a separate Chapter 3.

In this chapter the fundamentals of parallel computing are presented. It provides a general overview of the field of high performance computing and how performance of applications can be analyzed in this field.

2.2 Terms and Definitions

Before we go into further details the terms which are commonly used when dealing with parallel computing have to be defined. It is important to know the difference between processes and threads as they are the key figures in parallel computing.

Then, the difference between parallelism and concurrency is illustrated as they are often confused or said to be the same.

Eventually computational overhead and scalability are explained.

2.2.1 Process vs. Thread

Both threads and processes are methods of parallelizing an application. The difference between them is the way they are created and share resources.

A **process** is the execution of a list of statements (a sequential program). Processes have their own state information, use their own address space and interact with other processes only via an interprocess communication mechanism generally managed by the operating system. A master process may spawn subprocesses which are logically separated from the functionality of the master process and other subprocesses.

In contrast to processes, **threads** are typically spawned from processes for a short time to achieve a certain task and then terminate—**fork-join principle**. Within a process threads share the same state and same memory space, and can communicate with each other directly through shared variables.

2.2.2 Parallelism vs. Concurrency

Parallelism is physically simultaneous processing of multiple threads or processes with the objective to increase performance (this implies multiple pro-

cessing elements) whereas **concurrency** is logically simultaneous processing of threads or processes regardless of a performance gain (this does not imply multiple processing elements). I.e., concurrent execution of processes can be interleaved execution on a single processing element. Table 2.1 shows interleaved execution of 3 processes. In every cycle the process executed (X) changes: in the first cycle process 1 is executed, in the second cycle process 2 and after process 3 has started in cycle 3, process 1 continues its execution in cycle 4. Table 2.2 shows the execution of three parallel processes running on a computer with three processing elements.

Table 2.1: Interleaved concurrency

	Cycle								
Process	1	2	3	4	5	6	7	8	9
Process 1	X			X			X		
Process 2		X			X			X	
Process 3			X			X			X

Table 2.2: Parallelism

	Cycle								
Process	1	2	3	4	5	6	7	8	9
Process 1	X	X	X				X	X	X
Process 2			X	X	X	X	X	X	X
Process 3		X	X	X	X	X			

In general, concurrent programs can be executed in parallel whereas parallel programs cannot be reduced to logically concurrent processes.

For a description of concurrent programming languages and programming models see Gehani and McGettrick (1988).

2.2.3 Computational Overhead

An often used term when dealing with implementations of parallel programs is overhead.

Overhead is generally considered any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to be utilized or expanded to enable a particular goal (Wikipedia (2007b)).

In parallel computing this can be the excess of computation time to send data between one process to another.

Overhead may also be a criterion to decide whether a feature should be included or not in a parallel program or whether a parallel programming paradigm should be used or not.

2.2.4 Scalability

When using more than one processor to carry out a computation one is interested in the increase of performance which can be achieved by adding CPUs. The capability of a system to increase the performance under an increased load when resources (in this case CPUs) are added is referred to as **scalability**. For example, when the number of CPUs is doubled and the computation of the same task takes nearly half the time, the system scales well. If this is not the case because it only reduces time by 1 or 2 percent then bad scalability is given.

2.3 Computer Architecture

In this section computer architectures are briefly described. This knowledge is of major interest to understand how performance can be maximized when using parallel computing. First it is important to know how to make an application fast on a single processor. As soon as this is done one may begin to parallelize this application and think of how to supply data efficiently to more than one processors. In this section we try to summarize the key concepts in computer architecture and to show what we have to keep in mind when parallelizing code.

The beginning of this section deals with a taxonomy of the design alternatives for multiprocessors. After that a brief look into processor design and memory hierarchies is given. This section closes with a description of high performance computing servers.

2.3.1 Beyond Flynn's Taxonomy

Providing a high level standard for programming HPC applications is a major challenge nowadays. There is a variety of architectures for large-scale computing. They all have specific features and therefore there should be a taxonomy in which such architectures can be classified. About forty years ago, Flynn (1972) classified architectures on the presence of single or multiple either instructions or data streams known as Flynn's taxonomy:

Single Instruction Single Data (SISD) This type of architecture of CPUs (called uniprocessors), developed by the mathematician John

von Neumann, was the standard for a long time. These computers are also known as serial computers.

Multiple Instruction Single Data (MISD) The theoretical possibility of applying multiple instructions on a single datum is generally impractical.

Single Instruction Multiple Data (SIMD) A single instruction is applied by multiple processors to different data in parallel (data-level parallelism).

Multiple Instruction Multiple Data (MIMD) Processors apply different instructions on different data (thread-level parallelism). See Section 2.4 for details.

But these distinctions are insufficient for classifying modern computers according to Duncan (1990). There are e.g., pipelined vector processors capable of concurrent arithmetic execution and manipulating hundreds of vector elements in parallel.

Therefore, Duncan (1990) defines that a parallel architecture provides an explicit, high level framework for the development of parallel programming solutions by providing multiple processors that cooperate to solve problems through concurrent execution:

Synchronous architectures coordinate concurrent operations in lockstep through global clocks, central control unit, or vector unit controllers. These architectures involve pipelined vector processors (characterized by multiple, pipelined functional units, which implement arithmetic and Boolean operations), SIMD architectures (typically a control unit broadcasting a single instruction to all processors executing the instruction on local data) and systolic architectures (pipelined multiprocessors in which data flows from memory through a network of processors back to memory synchronized by a global clock).

MIMD architectures consist of multiple processors applying different instructions on local (different) data. The MIMD models are asynchronous computers although they may be synchronized by messages passing through an interconnection network (or by accessing data in a shared memory). The advantage is the possibility of executing largely independent sub calculations. These architectures can further be classified in distributed or shared memory systems whereas the former achieves interprocess communication via interconnection network and the latter via global memory each processor can address.

MIMD-based architectural paradigms involve MIMD/SIMD hybrids, dataflow architectures, reduction machines, and wavefront arrays. A MIMD architecture can be called a MIMD/SIMD hybrid if parts of the MIMD architecture are controlled in SIMD fashion (some sort of master/slave relation). In dataflow architectures instructions are enabled for execution as soon as all of their operands become available whereas in reduction architectures an instruction is enabled for execution when its results are required as operands for another instruction already enabled for execution. Wavefront array processors are a mixture of systolic data pipelining and asynchronous dataflow execution.

These paradigms have to be kept in mind when designing high performance software as each architecture implies a specific way of writing efficient code. Luckily, platform specific libraries and compilers tuned to run optimal on the corresponding hardware are available to developers. They help to reduce the amount of work necessary to create a parallel application.

2.3.2 Processor and Memory

A reduction of the execution time can be achieved when taking advantage of parallel computing and this is an essential possibility for improving performance. But a programmer can only achieve best performance when he is aware of the underlying hardware. Before making use of parallelism the performance of the sequential parts of the code has to be maximized. This means the serial routines of a parallel program have to be optimized to run on a single processor. In fact parallelism can be exploited at different levels starting with the processor itself (e.g., pipelining).

Furthermore, processors have to be provided with data sufficiently fast. If the memory is too slow having a fast processor does not pay. Typically memory which delivers good performance is too expensive to be economically feasible. This situation has led to the development of memory hierarchies.

Instruction Level Parallelism

A technique called **pipelining** allows a processor to execute different stages of different instructions at the same time. For example in the classic von Neumann architecture (SISD), processors complete consecutive instructions in a fetch-execute cycle: Each instruction is fetched, decoded, then data is fetched and the decoded instruction is applied on the data. A simple implementation where every instruction takes at most 5 clock cycles can be as follows (for details see Appendix A of Hennessy and Patterson (2007)).

Instruction fetch cycle (F) The current instruction is fetched from memory.

Instruction decode/register fetch cycle (D) Decode instructions and read registers corresponding to register source from the register file.

Execution/effective address cycle (X) The arithmetic logic unit (ALU) operates on the operands prepared in the prior cycle.

Memory access (M) Load or store instructions.

Write-back cycle (W) Write the result into the register file.

Pipelining allows parallel processing of these instructions. I.e., while one instruction is being decoded, another instruction may be fetched and so on. On each clock cycle simply a new instruction is started. Therefore after 5 cycles the pipe is filled and on each subsequent clock cycle an instruction is finished. The results of this execution pattern is shown in table 2.3.

Table 2.3: Pipelining

	Cycle								
Instruction Number	1	2	3	4	5	6	7	8	9
Instruction i	F	D	X	M	W				
Instruction $i+1$		F	D	X	M	W			
Instruction $i+2$			F	D	X	M	W		
Instruction $i+3$				F	D	X	M	W	
Instruction $i+4$					F	D	X	M	W

Pipelining only works if every instruction does not depend on its predecessor. If so, instructions may completely or partially be executed in parallel—instruction level parallelism. Pipelining can be exploited either in hardware or in software. In the hardware based approach (the market dominating approach) the hardware helps to discover and achieve parallelism dynamically. In the software based approach parallelism is exploited statically at compile time through the compiler.

Memory Hierarchies

The best way to make a program fast would be to take advantage entirely of the fastest memory available. But this would make the price of a computer extremely high. Memory hierarchies have become an economical solution in

a cost-performance sense. Between the processor and the main memory one or more levels of fast (smaller in size) memory is inserted—cache memory.

Programs tend to reuse data and instructions they have used recently. Therefore one wants to predict what instructions and data a program will use in the near future and keep them in higher (faster) levels of the memory hierarchy. This is also known as the **principle of locality** (Hennessy and Patterson (2007)). Fulfilling the principle of locality, cache memory stores the instructions and data which are predicted to be used next or in subsequent steps. This increases the throughput of the data provided that the predictions made were correct.

2.3.3 High Performance Computing Servers

High performance computing refers to the use of (parallel) supercomputers and computer clusters (see Section 2.1). This section gives an overview of high performance computing servers and how they are classified among all computer types.

Traditionally computers have been categorized as follows:

- personal computers
- workstations
- mini computers
- mainframe computers
- high performance computing servers (super computers)

This has changed rapidly in the last years, as the requirements of people are now different than they were before. Microprocessor-based computers dominate the market. PCs and workstations have emerged as major products in the computer industry. Servers based on microprocessors have replaced minicomputers and mainframes have almost been replaced with networks of multiprocessor workstations.

We are now interested in the last category namely high performance computing servers, because these computers have the highest computational power and they are mainly used for scientific or technical applications. The number of processors such a computer can have reaches from hundred to several thousand. The trend is to use 64 bit processors as this kind allows addressing more memory (and they have become commonplace now). This category of computers can be divided into the following categories: “vector computers” and “parallel computers”. The amount of memory they have

varies according to the type of these servers or applications which are running on them. The maximum amount of memory can absolutely be beyond several tera bytes.

This development started in the 1990's where parallel programming experienced major interest. With the increasing need of powerful hardware, computer vendors started to build supercomputers capable of executing more and more instructions in parallel. In the beginning vector supercomputing systems using a single shared memory were widely used to run large-scale applications. Due to the fact that vector systems were rather expensive in both purchasing and operation and bus based shared memory parallel systems were technologically bound to smaller CPU counts, the first distributed memory parallel platforms (DMPs) came up. The advantage of these platforms was that they were inexpensive with respect to their computational power and could be built in many different sizes. DMPs could be rather small like a few workstations connected via a network to form a cluster of workstations (COW) or arbitrarily large. In fact, individual organizations could buy clusters of a size that suited their budgets. The major disadvantage DMPs have is the higher latency and smaller throughput of messages passing through a network in comparison to a shared memory system. Although the technology has improved (Infiniband), there is still room for improvements.

Current HPC systems are still formed of vector supercomputers (providing highest level of performance, but only for certain applications), SMPs with two to over 100 hundred CPUs and DMPs. As more than one CPU has become commonplace in desktop machines, clusters can nowadays provide both small SMPs and big DMPs. When it comes to a decision which architecture to buy, companies, laboratories or universities often choose to use a cluster of workstations because the cost of networking has decreased as mentioned before and performance has increased since systems of this kind already dominate the Top500 list. Furthermore, COWs are so successful because they can be configured from commodity chips and network technology. Moreover they are typically set up with the open source Linux operating system and therefore offer high performance for a reasonable price.

New developments show that there are interesting alternatives. Some HPC platforms offer a global addressing scheme and therefore enable CPUs to directly access the entire system's memory. This can be done e.g., with ccNUMA (cache coherent nonuniform memory access) systems (for more information see e.g., Kleen (2005)) or global arrays (Nieplocha et al. (1996)). They have become an important improvement because they allow a processor to access local and remote memory with rather low latency.

Another advantage of having a global addressing scheme is that it enables compiler driven parallelizing techniques like OpenMP (see Section 2.5.4) to

be used on these systems. Otherwise the shared memory paradigm has to be maintained through a distributed shared memory subsystem like in Cluster OpenMP (Hoeflinger (2006)).

2.4 MIMD Computers

Recalling Section 2.3.1 we know that computer architecture plays an important role for designing parallel programs and from Section 2.3.3 we know how high performance computing servers have developed since the last decade. We are now interested in an architecture which supports parallel process execution and how distributed processing entities can be connected to form a single unit. According to Flynn's taxonomy this would be MIMD or SIMD architectures combined in a shared or distributed memory system.

On MIMD architectural based computers one has the highest flexibility in parallel computing. Whereas in SIMD based architectures a single instruction is applied to multiple data the MIMD paradigm allows to apply different instructions to different data at the same time. But SIMD architectures have become popular again recently. The Cell processor of IBM (Kahle et al. (2005)) is one of the promising CPUs of its kind. The focus in this thesis is on MIMD computers though.

There are two different categories in which such a system can be classified. The classification is based on how the CPU accesses memory. One possibility is to have distributed memory platforms (DMPs). Each processor has its own memory and when it comes to sharing of data, an interconnection network is needed. Exchange of data or synchronizing is done through message passing.

The other possibility is to have shared memory platforms (SMPs). All processors or computation nodes can access the same memory. The advantage is the better performance like in DMPs because messages do not have to be sent via a slow network. Each processor is able to fetch the necessary data directly from memory. But this kind of platform has a big limitation. The costs of having as much memory as a cluster of workstation would provide are enormous. That is why cluster or grid computing has become so popular.

A further possibility is to have some sort of a hybrid platform. On the one hand one has several shared memory platforms with a few computation cores on each and on the other hand they are connected via an interconnection network.

2.4.1 Shared Memory Systems

In a shared memory system multiple processors share one global memory. They are connected to this global memory mostly via bus technology. A big limitation of this type of machine is that though they have a fast connection to the shared memory, saturation of the bandwidth can hardly be avoided. Therefore scaling shared memory architectures to more than 32 processors is rather difficult. Recently this type of architecture has gained more attention again. Consumer PCs with two to four processors have become common since the last year. High-end parallel computers use 16 up to 64 processors. Recent developments try to achieve easy parallelization of existing applications through parallelizing compilers (e.g., OpenMP see Section 2.5.4).

2.4.2 Distributed Memory Systems

Since the developments done by the NASA in the 1990's (the Beowulf project, Becker et al. (1995)), this kind of MIMD architecture has become really popular. Distributed memory systems can entirely be built out of stock hardware and therefore give access to cheap computational power. Furthermore, a big advantage is that this architecture can easily be scaled up to several hundreds or thousands of processors. A disadvantage of this type of systems is that communication between the nodes is achieved through common network technology. Therefore sending messages is rather expensive in terms of latency and bandwidth. Another disadvantage is the complexity of programming parallel applications. There exist several implementations or standards which assist the developer in achieving message passing (for MPI see Section 2.5.2 or PVM see Section 2.5.3).

Beowulf Clusters

In 1995 NASA scientist completed the Beowulf project, which was a milestone toward cluster computing.

"Beowulf Clusters are scalable performance clusters based on commodity hardware, on a private system network, with open source software (Linux) infrastructure. The designer can improve performance proportionally with added machines. The commodity hardware can be any of a number of mass-market, stand-alone compute nodes as simple as two networked computers each running Linux and sharing a file system or as complex as 1024 nodes with a high-speed, low-latency network." (from `http://www.beowulf.org/`)

Since then building an efficient low-cost distributed memory computer has not been difficult anymore. Indeed it has become the way to build MIMD

computers.

The Grid

Another (new) type of achieving high performance computing is to use a grid (Foster and Kesselman (1999)). In a grid, computers which participate are connected to each other via the Internet (or other wide area networks). These distributed resources are shared via a grid software. The software monitors separate resources and if necessary supplies them with jobs. The research project SETI@home (Korpela et al. (2001) and Anderson et al. (2002)) can be said to be the first grid-like computing endeavor. Over 3 million users work together in processing data acquired by the Acribo radio telescope, searching for extraterrestrial signals. Most of the power is gained by utilizing unused computers (e.g., with a screen saver).

The idea behind it is that computers connected to a grid can combine their computational power with other available computing resources in the grid. Then, researchers or customers are able to use the combined resources for their projects. The aim of development of grid computing is to find standards so that grid computing becomes popular not only in academic institutes but also in industry. Today there are several initiatives in Europe. For example D-Grid (`http://www.d-grid.de`) funded by by BMBF, the Federal Ministry of Education and Research of Germany or the Austrian Grid (`http://www.austriangrid.at`) funded by the bm.w_f, the Federal Ministry for Science and Research of Austria.

In a grid more attention has to be paid on heterogeneous network computing than in smaller DMPs which typically consist of many machines with the same architecture and are most likely from the same vendor. Computers connected to a grid are made from different vendors, have different architectures and different compilers. All in all they may differ in the following areas:

- architecture
- computational speed
- machine load
- network load and performance
- file system
- data format

A Grid middleware software like the open source software "globus toolkit" (Foster and Kesselman (1997)—software available from `http://www.globus.org/`) interconnects all the (heterogeneous) distributed resources in a network. All interfaces between these heterogeneous components have to be standardized and thus enable full interoperability between them.

2.5 Parallel Programming Models

In Section 2.3.3 and 2.4 we presented the hardware used when dealing with parallel computing which is more or less given to the developers. The major task for a programmer is to use the given infrastructure and provide efficient solutions for computational intensive applications.

Creating a parallel application is a challenging task for every programmer. They have to distribute all the computations involved to a large number of processors. It is important that this is done in a way so that each of these computation nodes performs roughly the same amount of work. Furthermore, developers have to ensure that data required for the computation is available to the processor with as little delay as possible. Therefore some sort of coordination is required for the locality of data in the memory (for more information on the design of a parallel program see Section 2.6). One might think that this would be hard work, and indeed it is. But there are programming models for high performance parallel computing already available which make the developer's life easier.

When parallel computing came up parallel programming languages like Concurrent C, Ada, or Linda (see e.g., Gehani and McGettrick (1988) and Wilson and Lu (1996)) had been developed. Only few of them experienced any real use. There have also existed many implementations of computer vendors for their own machines. For a long time there was no standard in sight as no agreement between hardware vendors and developers emerged. It was common that application developers had to write separate HPC applications for each architecture.

In the 1990's broad community efforts produced the first defacto standards for parallel computing. Commonalities were identified between all the implementations available and the field of parallel computing had been understood better. Since then libraries and concrete implementations as well as compilers capable of automatic parallelization have been developed.

The programming models presented in this section are common and widely used. First we start with a description of fundamental concepts of message passing as they are needed in subsequent sections. Then the first generation, the Message Passing Interface (MPI—see Message Passing In-

terface Forum (1994)) and Parallel Virtual Machine (PVM—see Geist et al. (1994)), are described. The second generation of HPC programming models involve OpenMP (see OpenMP Architecture Review Board (2005)) among others. These programming models are not the only ones but are, as mentioned above, commonly used.

2.5.1 Fundamentals of Message Passing

To understand subsequent sections it is important to know the exact meaning of message passing and the existing building blocks. On a distributed memory platform (Section 2.4.2) data has to be transmitted from one computation node to the other to carry out computations in parallel. The most commonly used method for distributing data among the nodes is message passing.

A message passing function is a function which explicitly transmits data from one process to another. With these functions, creating parallel programs can be extremely efficient. Developers do not have to care about low level message passing anymore.

To identify a process in a communication environment message passing libraries make use of identifiers. In MPI they are called "ranks" and in PVM "TaskIDs".

Often used concepts in message passing are buffering, blocking or non-blocking communication, load balancing and collective communication.

Buffering

Communication can be buffered which means that e.g., the sending process can continue execution and need not wait for the receiving process signaling it is ready to receive. Otherwise one would have a synchronous communication.

Blocking and Non-blocking Communication

A communication is blocking if a receiving process has to wait if the message from the sending process is not available. That means the receiving process remains idle until the sending process starts sending. In non-blocking communication the receiving process sends a request to another process and continues executing. At a later time the process checks if the message has arrived in the meantime.

Load Balancing

To distribute data as evenly as possible among the processes, load balancing is a good method. This can be achieved either through data decomposition (identical programs or functions operate on different portions of the data) or function decomposition (different programs or functions perform different operations). For more details on workload allocation see Section 4.2 in Geist et al. (1994).

Collective Communication

In a parallel program we do not want that only one process does most of the work. This is an issue if it comes to sending of data. Assume that all of the data is available to one process (A) and it wants to send it to all of the other. The last receiving process remains idle until process A has sent data to all of the other processes. This can be avoided when involving more than one process in sending of data. This is called a broadcast. A broadcast is a collective communication in which a single process sends the same data to every other process. Depending on the topology of the system there is an optimized broadcast available (e.g., tree-structured communication). See Chapter 5 in Pacheco (1997) for more details on collective communication in MPI.

2.5.2 The Message Passing Interface (MPI)

In 1994 the Message Passing Interface Forum (MPIF) has defined a set of library interface standards for message passing. Over 40 organizations participated in the discussion which started in 1993. The aim was to develop a widely accepted standard for writing message passing programs. The standard was called the Message Passing Interface (see Message Passing Interface Forum (1994)).

As mentioned in Section 2.5 there were lots of different programming models and languages for high performance computing available. With MPI a standard was established that should be practical, portable, efficient and flexible. After releasing Version 1.0 in 1994 the MPIF continued to correct errors and made clarifications in the MPI document so that in June 1995 Version 1.1 of MPI was released. Further corrections and clarifications have been made to version 1.2 of MPI and with Version 2 completely new types of functionality were added (see Message Passing Interface Forum (2003) for details).

MPI is now a library providing higher level routines and abstractions built

upon lower level message passing routines which can be called from various programming languages like C or Fortran. It contains all the infrastructure for inter-process communication. Last but not least an interface wrapper is available to R. MPI communication can be set up using the R extension **Rmpi** (see Section 3.3 for details).

Compared to other parallel programming MPI has the following advantages and disadvantages:

Advantages

- portability through clearly defined sets of routines—a lot of implementations are available for many different platforms
- better performance compared to PVM as MPI's implementations are adapted to the underlying architecture.
- interoperability within the same implementation of MPI
- dynamic process creation since MPI-1.2
- good scalability

Disadvantages

- MPI is not a self contained software package
- development of parallel programs is difficult
- no resource management like PVM
- heterogeneity only within an architecture

For a detailed comparison of PVM and MPI see Geist et al. (1996) and Gropp and Lusk (2002).

Implementations of the MPI Standard

As MPI is only a standard, specific implementations are needed to use message passing routines as library calls in programming languages like C or FORTRAN. There has been huge effort to provide good implementations which are widely used. This is because of the fact that the primary goal was portability. A lot of different platforms are now supported. What follows is a description of commonly used and well known MPI implementations.

LAM/MPI is an open source implementation of MPI. It runs on a network of UNIX machines connected via a local area network or via the Internet. On each node runs a UNIX daemon which is a micro-kernel plus a few system processes responsible for network communication among other things. The micro-kernel is responsible for the communication between local processes (see Burns et al. (1994)). The LAM/MPI implementation includes all of the MPI-1.2 but not everything of the MPI-2 standard. LAM/MPI is available at `http://www.lam-mpi.org/`.

MPICH is a freely available complete implementation of the MPI specification (MPICH1 includes all MPI-1.2 and MPICH2 all MPI-2 specifications). The main goal was to deliver high performance with respect to portability. "CH" in MPICH stands for "Chameleon" which means adaptability to one's environment and high performance as chameleons are fast (see Gropp et al. (1996)). MPICH had been developed during the discussion process of the MPIF and had immediately been available to the community after the standard had been confirmed. There are many other implementations of MPI based on MPICH available because of its high and easy portability. Sources and Windows binaries are available from `http://www-unix.mcs.anl.gov/mpi/mpich/`.

Open MPI has been built upon three MPI Implementations like LAM/MPI among the others. It is a new open source MPI-2 Implementation. The goal of Open MPI is to implement the full MPI-1.2 and MPI-2 specifications focusing on production-quality and performance (see Gabriel et al. (2004)). Open MPI can be downloaded from `http://www.open-mpi.org/`.

For setting up the LAM/MPI implementation in Debian GNU Linux see Appendix A. For configuring the environment and running jobs on a cluster see Appendix B.

2.5.3 Parallel Virtual Machine (PVM)

The PVM system is another implementation of a functionally complete message passing model. It is designed to link computing resources for example in a heterogenous network and to provide developers with a parallel platform. As the message passing model seemed to be the paradigm of choice in the late 80's the PVM project was created in 1989 to allow developers to exploit distributed computing across a wide variety of computer types. A key concept in PVM is that it makes a collection of computers appear as one large *virtual* machine, hence its name (Geist et al. (1994)). The PVM system can

be found on `http://www.csm.ornl.gov/pvm/`. How to setup this system on a Linux machine is explained in Appendix A. How to start a PVM job on a cluster is shown in Appendix B.

PVM has like other message passing implementations its advantages and disadvantages:

Advantages

- virtual machine concept
- good support of heterogeneous networks of workstations
- high portability—available for many different platforms
- resource management, load balancing and process control
- self-contained software package
- dynamic process creation
- support for fault tolerant programming
- good scalability

Disadvantages

- more overhead because of portability
- development of parallel programs is difficult
- not as flexible as MPI as PVM is a specific implementation
- not as much message passing functionality as MPI has

2.5.4 OpenMP

A completely different approach in the parallel programming model context is the use of shared memory. Message passing is built upon this shared memory model which means that every processor has direct access to the memory of every other processor in the system (see Section 2.4.1). This class of multiprocessor is also called Scalable Shared Memory Multiprocessor (SSMP).

Like the development of the MPI specifications the development of OpenMP started for one simple reason: portability. Prior to the introduction of OpenMP as an industry standard every vendor of shared memory systems

created its own proprietary extension for developers. Therefore a lot of people interested in parallel programming used portable message passing models like MPI (see Section 2.5.2) or PVM (see Section 2.5.3). As there was an increasing demand for a simple and scalable programming model and the desire to begin parallelizing existing code without having to completely rewrite it, the OpenMP Architecture Review Board released a set of compiler directives and callable runtime library routines known as the OpenMP API (see OpenMP Architecture Review Board (2005)). This directives and routines extend the programming languages FORTRAN and C or C++ respectively to express shared memory parallelism.

Furthermore this programming model is based on the fork/join execution model which makes it easy to get parallelism out of a sequential program. Therefore unlike in message passing the program or algorithm need not to be completely decomposed for parallel execution. Given a working sequential program it is not difficult to incrementally parallelize individual loops and thereby realize a performance gain in a multiprocessor system (Dagum (1997)). The specification and further information can be found on the following website: `http://www.openmp.org/`.

OpenMP has a major advantage—it is easy to use. Further advantages and disadvantages are:

Advantages

- writing parallel programs with OpenMP is easy—incrementally parallelizing of sequential code is possible
- dynamic creation of threads
- unified code—if OpenMP is not available directives are treated as comments

Disadvantages

- the use of OpenMP depends on the availability of a corresponding compiler
- a shared memory platform is needed for efficient programs
- a lot more overhead compared to message passing functions because of implicit programming
- scalability is limited by the architecture

2.6 Design of Parallel Programs

In this section design issues when dealing with parallel programming are presented.

First, the granularity of parallel programs plays an important role when designing a parallel application although the grain size is to some extent limited by the architecture. It is a classification of the numbers of instructions performed in parallel.

Second, a fundamental differentiation of parallel programs (particularly on MIMD platforms) is explained. Programs are categorized based on the organization of the computing tasks.

Eventually, this section concludes with a differentiation of how parallelism can be achieved. It is a summary of the three fundamental approaches of writing parallel programs.

2.6.1 Granularity of Parallel Programs

Having the parallel hardware available, the developer has to be aware of the software like the available compilers and libraries (parallel programming models) to use for parallel computations.

What goes hand in hand with the decision for a parallel programming model on a specific architecture is the granularity of the resulting parallelism. Granularity is in this case referred to as the number of instructions performed in parallel. This reaches from fine-grained (instruction level) parallelism to coarse-grained (task or procedure level) parallelism:

Levels of Parallelism

Instruction level parallelism single instructions are processed concurrently.

Data level parallelism the same instruction is processed on different data at the same time.

Loop level parallelism a block instruction from a loop can be run for each loop iteration in parallel.

Task level parallelism larger program parts (mostly program procedures) are run in parallel.

With choosing a specific grain size the developer has to be aware of the following tradeoff: a fine-grained program provides considerable flexibility

with respect to load balancing, but it also involves a high overhead of processing time due to a larger management need and communication expense (Sevcikova (2004)).

Furthermore, the available hardware architecture has to be taken into consideration. Fine-grained parallel programs would not benefit much in a distributed memory system. Indeed the cost for transferring data over the network would be too high with respect to the runtime of the instruction itself. In contrast coarse grained parallelism would perform better, as communication is held at a minimum as larger program parts are run in parallel.

2.6.2 Organization of Parallel Tasks

Parallel programming paradigms can be classified according to the process structure. According to Geist et al. (1994) parallel programs can be organized in three different programming approaches:

Crowd computation is if a collection of closely related processes, typically executing the same code, perform computations on different portions of the workload. This paradigm can be further subdivided into

- *Master-slave* or host-node model in which one process (the master) is responsible for process spawning (this means creation of a new process), initialization, collection and display of results. Slave programs perform the actual computation involved.
- *Node-only* model have no master process but autonomous processes from which one takes over the non-computational parts in addition to the computation itself.

Tree computations exhibit a tree-like process control structure. It can also be seen as a tree-like parent-child relationship.

Hybrid is a combination of the crowd and the tree model.

Furthermore, parallel programs can be further classified either as the single program multiple data (SPMD) or multiple program multiple data paradigm (MPMD).

Single program multiple data means that there is only one program available to all computation nodes. Data is distributed equally amongst all nodes in the program (**data decomposition**). Then each node applies the same program on its own part of the data. One should not confuse

SPMD with the SIMD paradigm as in the latter case low level instructions (directly from the CPU) are applied to previously distributed data and therefore is another form of programming.

Multiple program multiple data is a thread like paradigm. On each node it is possible to run different programs with either the same or different data (**function decomposition**). Each program forms a separate functional unit. This is what one would assume when speaking of MIMD architectures but both models (SPMD and MPMD) form the MIMD programming paradigm.

2.6.3 Types of Achieving Parallelism

Providing parallelism to a program can be achieved with one of the following approaches (Sevcikova (2004)). They are different in the complexity of writing parallel programs.

Implicit parallelism is writing a sequential program and using an advanced compiler to create parallel executables. The developer does not need to know how a program has to be parallelized. This involves a lot of overhead as automated parallelization is difficult to achieve.

Explicit parallelism with implicit decomposition is writing a sequential program and marking regions for parallel processing. In this case the compiler does not need to analyze the code as the developer defines which regions are to be parallelized. With this type of parallelism not as much overhead as with implicit parallelism is involved. This can be achieved via extensions to standard programming languages (e.g., OpenMP).

Explicit parallelism is writing explicit parallel tasks and handling communication between processes manually. Overhead is only produced through communication and thus offer the possibility of creating very efficient code. This type of parallelism is achieved through parallel programming libraries like MPI or PVM.

With the information given in this section we now know the design issues of parallel programs. The next step is to analyze performance of parallel applications.

2.7 Performance Analysis

In high performance computing one of the most important methods of improving performance is taking advantage of parallelism. As explained in Section 2.3.2 there are other possibilities to make a program run fast like following the principle of locality or making use of pipelining. This is somehow architecture dependent and therefore developers should try to parallelize their code. A rule of thumb is to make the frequent case fast, this often involves parallelizing of this case. How these improvements can be analyzed is discussed in this section.

2.7.1 Execution Time and Speedup

The best way to analyze the performance of an application is to measure the execution time. Consequently an application can be compared with an improved version through the execution times. The performance gain can then be expressed as shown in Equation 2.1.

$$Speedup = \frac{t_s}{t_e} \tag{2.1}$$

where

t_s denotes the execution time for a program without enhancements (serial version)

t_e denotes the execution time for a program using the enhancements (enhanced version)

2.7.2 Amdahl's Law

If we have a program running on a classical von Neumann machine with a total execution time t_s and the possibility to port this program to a multiprocessor system with p processors we would expect that the parallel execution time t_p would be as in Equation 2.2

$$t_p = \frac{t_s}{p} \tag{2.2}$$

and consequently the speedup would be equal to the number of processors p. But this is an ideal scenario. Linear speedup can hardly be achieved because not every part of a program can be parallelized in the same way. In general different parts of an application are executed with different speeds and/or use different resources.

To find an estimate of good quality the program has to be separated into its different parts which in turn can be subject to enhancements. Amdahl (1967) made significant initial work on this topic.

If we assume that a fraction f of an algorithm can be ideally parallelized using p processors whereas the remaining fraction $1-f$ cannot be parallelized the total execution time of the enhanced algorithm would be as shown in Equation 2.3.

$$t_p = f\frac{t_s}{p} + (1-f)t_s = \frac{t_s(f+(1-f)p)}{p} \tag{2.3}$$

The speedup of an algorithm that results from increasing the speed of its fraction f is inversely proportional to the size of the fraction that has to be executed at the slower speed (interpretation from Section 1.5 of Kontoghiorghes (2006)).

Equation 2.4 shows the corresponding speedup that can be achieved. If f equals one (the program can be ideally parallelized) the speedup would equal the amount of processors p. If f is zero (the program cannot be parallelized) the speedup would be one.

$$speedup = \frac{p}{f+(1-f)p} \tag{2.4}$$

Normally, as $f < 1$ inequality 2.5 is true. It has became known as *Amdahl's Law for parallel computing*. Figure 2.7.2 shows the speedup possible for 1 to 10 processors if the fraction of parallelizable code is 100% (linear speedup), 90%, 75% and 50%.

$$speedup < \frac{1}{1-f} \tag{2.5}$$

2.8 Hardware and Software Used

2.8.1 High Performance Computing Servers

The Hardware used for the applications presented in this thesis is going to be described in this section.

At the Vienna University of Economics and Business Administration (WU) a research institute called Research Institute for Computational Methods (Forschungsinstitut für rechenintensive Methoden or FIRM for short) hosts a cluster of Intel workstations, known as cluster@WU. For more information on the research institute or on the cluster visit `http://www.wu-wien.ac.at/firm`.

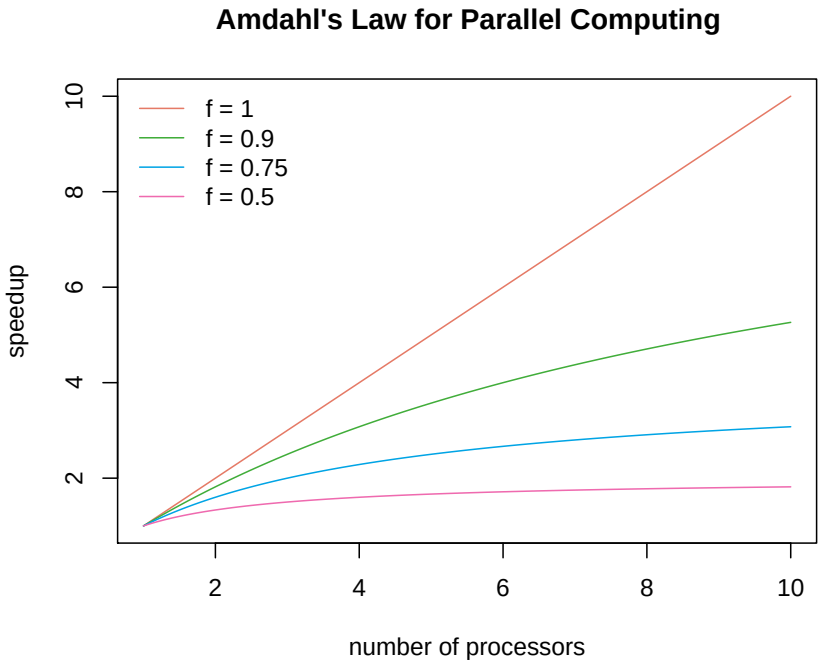

Figure 2.1: Amdahl's Law for parallel computing

Furthermore, for experimenting and interactive testing an AMD Opteron server with four computation cores is available at the Department of Statistics and Mathematics.

Cluster@WU

All programs have been tested as well as benchmarked on this cluster of workstations running the resource management system Sun Grid Engine (Gentzsch et al. (2002) and Sun Microsystems, Inc. (2007)—see also Appendix B for more information on the grid engine). With a total of 152 64-bit computation nodes and a total of 336 gigabytes of RAM, Cluster@WU is by now amongst the fastest supercomputers in Austria.

The cluster consists of four workstation with four cores each and 16 gigabytes of RAM. They are called "bignodes" as they offer more power to the grid user. The queue for running applications on these bignodes is called `bignode.q`. If a shared memory program is to be run (e.g., an OpenMP program), bignodes are the computers of choice. The other nodes consist of dual core CPUs and less memory. They are combined in the queue `node.q`. Table 2.4 provides detailed information about the specs of the queues.

AMD Opteron Server

The Opteron machine has four cores which have access to a total of 12 gigabytes of shared memory. The specification of this machine shows table 2.5.

2.8.2 Utilized Software

The software presented in this section is used for developing and running code of this thesis. The operating system is Debian GNU Linux (`http://www.debian.org`).

Sun Grid Engine is an open source cluster resource management and scheduling software. On cluster@WU version 6.0 of the Grid Engine manages the remote execution of cluster jobs. It can be obtained from `http://gridengine.sunsource.net/`, the commercial version Sun N1 Grid Engine can be found on `http://www.sun.com/software/gridware/`. See Appendix B how to use the Grid Engine with parallel environments.

Compiler used in this thesis are the Intel C++ Compiler (Intel Corporation (2007a)) the Intel Fortran Compiler (Intel Corporation (2007b)) both in version 9.1 and the GNU Compiler Collection 4.1.2 (Stallman and the GCC Developer Community (2007)). For compiling OpenMP code the Intel compiler is being used since the GNU compiler supports these parallelizing techniques as of version 4.2 or later.

Table 2.4: cluster@WU specification

bignode.q – 4 nodes	
2	Dual Core Intel XEON CPU 5140 @ 2.33 GHz
16	GB RAM
node.q – 68 nodes	
1	Intel Core 2 Duo CPU 6600 @ 2.4 GHz
4	GB RAM

Table 2.5: Opteron server specification

Opteron server – 1 machine	
2	Dual Core AMD Opteron @ 2.4 GHz
12	GB RAM
1.2	TB RAID 5 storage

R is a free software environment for Statistical Computing and Graphics (R Development Core Team (2007b)). R-patched 2.5.1 is used in this thesis. There is a separate chapter about R and high performance computing (Chapter 3).

PVM version 3.4 is used for running the corresponding programs (see Section 2.5.3).

LAM/MPI version 7.1.3 is used for running MPI programs (see Section 2.5.2).

2.9 Conclusion

This chapter provided an overview of the state of the art in parallel computing. The ongoing development of computer architectures shows that parallel computing is becoming increasingly important as the newest commodity processors are already equipped with two to four computation cores. At the same time in the high performance computing segment a further substantial increase in parallel computational power is taken place. An interesting development certainly is grid computing. Sharing unused resources and on the other hand having access to a tremendous amount of computational power may be the approach for large-scale computing in the future. Increasing network bandwidth and a growing amount of unused computational resources makes this possible.

Tools for creating programs exist for every kind of platform. The most promising of them is certainly OpenMP which may be the paradigm of choice for the mainstream. A lack in performance and the limitation to shared memory platforms compared to message passing environments has to be considered though. MPI and PVM are still commonly used as they deliver highest performance on many different platforms through their portability. A major disadvantage is the complexity of writing programs using message passing. Years of experience are needed to write efficient algorithms. Nevertheless, it remains to be an interesting challenge.

Chapter 3

High Performance Computing and R

3.1 Introduction

This chapter provides an overview of the capabilities of R (R Development Core Team (2007b)) in the area of high performance computing. A short description of the software package R is given at the beginning of this chapter. Subsequently extensions to the base environment (so called packages) which provide high performance computation functionality to R are going to be explained. Among these extensions there is the package called **paRc**, which was developed in the course of this thesis.

Examples shown in this chapter have been produced on cluster@WU (see Section 2.8 for details).

3.2 The R Environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. R is open source originally developed by Ross Ihaka and Robert Gentleman (Ihaka and Gentleman (1996)). Since 1997 a group of scientists (the "R Core Team") is responsible for the development of the R-project and has write access to the source code. R has a homepage which can be found on `http://www.R-project.org`. Sources, binaries and documentation can be obtained from CRAN, the Comprehensive R Archive Network (`http://cran.R-project.org`). Among other things R has (R Development Core Team (2007b))

- an effective data handling and storage facility,

- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either directly at the computer or on hardcopy, and
- a well developed, simple and effective programming language (called 'R') which includes conditionals, loops, user defined recursive functions and input and output facilities. (Indeed most of the system supplied functions are themselves written in the R language.)

R is not only an environment for statistical computing and graphics but also a freely available high-level language for programming. It can be extended by standardized collections of code called "packages". So developers and statisticians around the world can participate and provide optional code to the base R environment. Developing and implementing new methods of data analysis can therefore be rather easy to achieve (for more information about R and on how R can be extended see Hornik (2007) and R Development Core Team (2007a)).

Now, as datasets grow bigger and bigger and algorithms become more and more complex, R has to be made ready for high performance computing. Indeed, R is already prepared through a few extensions explained in the subsequent chapters. The packages mentioned in this chapter can be obtained from CRAN except **paRc** which can be obtained from R-Forge (`http://R-Forge.R-project.org`), a platform for collaborative software development for the R community (Theussl (2007a)).

3.3 The Rmpi Package

The Message Passing Interface (MPI) is a set of library interface standards for message passing and there are many implementations using these standards (see also Section 2.5.2). **Rmpi** is an interface to MPI (Yu (2002) and Yu (2006)). As of the time of this writing **Rmpi** uses the LAM implementation of MPI. For process spawning the standard MPI-1.2 is required which is available in the LAM/MPI implementation as LAM/MPI (version 7.1.3) supports a large portions of the MPI-2 standard. This is necessary if one likes to use interactive spawning of R processes. With MPI versions prior to MPI-1.2 separate R processes have to be created by hand using `mpirun` (part of many MPI implementations) for example.

Rmpi contains a lot of low level interface functions to the MPI C-library. Furthermore, a handful of high level functions are supplied. A selection of routines is going to be presented in this section.

A windows implementation of this package (which uses MPICH2) is available from `http://www.stats.uwo.ca/faculty/yu/Rmpi`.

3.3.1 Initialization and Status Queries

The LAM/MPI environment has to be booted prior to using any message passing library functions. One possibility is to use the command line, the other is to load the **Rmpi** package. It automatically sets up a (small—1 host) LAM/MPI environment (if the executables are in the search path).

When using the Sun Grid Engine (SGE) or other queueing systems to boot the LAM/MPI parallel environment the developer is not engaged with setting up and booting the environment anymore (see Appendix B on how to do this). On a cluster of workstations this is the method of choice.

Management and Query Functions

`lamhosts()` finds the hostname associated with its node number.

`mpi.universe.size()` returns the total number of CPUs available to the MPI environment (i.e., in a cluster or in a parallel environment started by the grid engine).

`mpi.is.master()` returns TRUE if the process is the master process or FALSE otherwise.

`mpi.get.processor.name()` returns the hostname where the process is executed.

`mpi.finalize()` cleans all MPI states (this is done when calling `mpi.exit` or `mpi.quit`.

`mpi.exit()` terminates the mpi communication environment and detaches the **Rmpi** package which makes reloading of the package **Rmpi** in the same session impossible.

`mpi.quit()` terminates the mpi communication environment and quits R.

Example 3.1 shows how the configuration of the parallel environment can be obtained. First it returns the hosts connected to the parallel environment and then prints the number of CPUs available in it. After a query if this

process is the master process, the hostname the current process runs on is returned.

Example 3.1 Queries to the MPI communication environment

```
> library("Rmpi")
> lamhosts()

node023 node023 node037 node037 node017 node017 node063 node063
      0       1       2       3       4       5       6       7


> mpi.universe.size()

[1] 8


> mpi.is.master()

[1] TRUE


> mpi.get.processor.name()

[1] "node023"
```

3.3.2 Process Spawning and Communication

In **Rmpi** it is easy to spawn R slaves and use them as workhorses. The communication between all the involved processes is carried out in a so called communicator (`comm`). All processes within the same communicator are able to send or receive messages from other processes. The processes are identified through their commrank (see also the fundamentals of message passing in Section 2.5.1). The big advantage of **Rmpi** slaves is, that they can be used interactively when using the default R slave script.

Process Management Functions

`mpi.spawn.Rslaves(nslaves = mpi.universe.size(), ...)` spawns a number (`nslaves`) of R workhorses to those hosts automatically chosen by MPI. For other arguments represented by ... to this function we refer to Yu (2006).

`mpi.close.Rslaves(dellog = TRUE, ...)` closes previously spawned slaves and returns 1 if successful.

`mpi.comm.size()` returns the total number of members in a communicator.

`mpi.comm.rank()` returns the rank (identifier) of the process in a communicator.

`mpi.remote.exec(cmd, ..., ret = TRUE)` executes a command `cmd` on R slaves with ... arguments to `cmd` and returns executed results if `ret` is `TRUE`.

In Example 3.2 as many slaves are spawned as are available in the parallel environment. The size of the communicator is returned (1 master plus the spawned slaves) and a remote query of the commrank is carried out. Before the slaves are closed the commrank of the master is printed.

Example 3.2 Process management and communication

```
> mpi.spawn.Rslaves(nslaves = mpi.universe.size())
        8 slaves are spawned successfully. 0 failed.
master (rank 0, comm 1) of size 9 is running on: node023
slave1 (rank 1, comm 1) of size 9 is running on: node023
slave2 (rank 2, comm 1) of size 9 is running on: node023
slave3 (rank 3, comm 1) of size 9 is running on: node037
slave4 (rank 4, comm 1) of size 9 is running on: node037
slave5 (rank 5, comm 1) of size 9 is running on: node017
slave6 (rank 6, comm 1) of size 9 is running on: node017
slave7 (rank 7, comm 1) of size 9 is running on: node063
slave8 (rank 8, comm 1) of size 9 is running on: node063

> mpi.comm.size()
[1] 9

> mpi.remote.exec(mpi.comm.rank())
  X1 X2 X3 X4 X5 X6 X7 X8
1  1  2  3  4  5  6  7  8

> mpi.comm.rank()
[1] 0

> mpi.close.Rslaves()
[1] 1
```

3.3.3 Built-in High Level Functions

Rmpi provides many high level functions. We selected a few of them which we think are commonly used. Most of them have been utilized to build parallel programs presented in subsequent chapters.

High Level Functions

`mpi.apply(x, fun, ...)` applies a function `fun` with additional arguments `...` to a specific part of a vector `x`. The return value is of type list with the same length as of `x`. The length of `x` must not exceed the number of R slaves spawned as each element of the vector is used exactly by one slave. To achieve some sort of load balancing one can use the corresponding apply functions below.

`mpi.applyLB(x, fun, ...)` applies a function `fun` with additional arguments `...` to a specific part of a vector `x`. There are a few more variants explained in Yu (2006).

`mpi.bcast.cmd(cmd = NULL, rank = 0, ...)` broadcasts a command `cmd` from the sender `rank` to all R slaves and evaluates it.

`mpi.bcast.Robj(obj, rank = 0, ...)` broadcasts an R object `obj` from process rank `rank` to all other processes (master and slaves).

`mpi.bcast.Robj2slave(obj, ...)` broadcasts an R object `obj` to all R slaves from the master process.

`mpi.parSim( ... )` carries out a Monte Carlo simulation in parallel. For details on this function see the package manual (Yu (2006)) and on Monte Carlo simulation the applications in Chapter 5.

How to use the high level function `mpi.apply()` is shown in Example 3.3. A vector of n random numbers is generated on each of the n slaves and are returned to the master as a list (each list element representing one row). Finally a $n \times n$ matrix is formed and printed. The output of the matrix shows for each row the same random numbers. This is because of the fact, that each slave has the same seed. This problem can be solved by using packages like **rsprng** or **rlecuyer** which provide parallel random number generation (for more information see the descriptions of the packages in Section 3.7.

Example 3.3 Using mpi.apply

```
> n <- 8
> mpi.spawn.Rslaves(nslaves = n)
```

```
        8 slaves are spawned successfully. 0 failed.
master (rank 0, comm 1) of size 9 is running on: node023
slave1 (rank 1, comm 1) of size 9 is running on: node023
slave2 (rank 2, comm 1) of size 9 is running on: node023
slave3 (rank 3, comm 1) of size 9 is running on: node037
slave4 (rank 4, comm 1) of size 9 is running on: node037
slave5 (rank 5, comm 1) of size 9 is running on: node017
slave6 (rank 6, comm 1) of size 9 is running on: node017
slave7 (rank 7, comm 1) of size 9 is running on: node063
slave8 (rank 8, comm 1) of size 9 is running on: node063

> x <- rep(n, n)
> rows <- mpi.apply(x, runif)
> X <- matrix(unlist(rows), ncol = n, byrow = TRUE)
> round(X, 3)
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]
[1,] 0.621 0.209 0.038 0.319 0.595 0.437 0.034 0.344
[2,] 0.621 0.209 0.038 0.319 0.595 0.437 0.034 0.344
[3,] 0.621 0.209 0.038 0.319 0.595 0.437 0.034 0.344
[4,] 0.621 0.209 0.038 0.319 0.595 0.437 0.034 0.344
[5,] 0.621 0.209 0.038 0.319 0.595 0.437 0.034 0.344
[6,] 0.621 0.209 0.038 0.319 0.595 0.437 0.034 0.344
[7,] 0.621 0.209 0.038 0.319 0.595 0.437 0.034 0.344
[8,] 0.621 0.209 0.038 0.319 0.595 0.437 0.034 0.344

> mpi.close.Rslaves()
[1] 1
```

3.3.4 Other Important Functions

To complete the set of important functions supplied by the **Rmpi** package the following functions have to be explained.

Collective Communication Routines

`.mpi.gather(x, type, rdata, root = 0, ...)` gathers data distributed on the nodes (`x`) to a specific process (mostly the master) into a single array or list (depending on the length of the data) of type `type` which can be integer, double or character. It performs a send of messages from each member in a communicator. A specific process (`root`) accumulates this messages into a single array or list prepared with the `rdata` command.

`.mpi.scatter(x, type, rdata, root = 0, ...)` sends to each member of a communicator a partition of a vector `x`, type `type` which can be integer, double or character, from a specified member of the group (mostly the master). Each member of the communicator receive its part of `x` after preparing the receive buffer with the argument `rdata`.

3.3.5 Conclusion

The package **Rmpi** implements many of the routines available in MPI-2. But there are routines that have to be omitted or are not included because they are not needed for use with R (e.g., data management routines are not necessary as R has its own tools for data handling). A really interesting aspect of the **Rmpi** package is the possibility to spawn interactive R slaves. That enables the user to interactively define functions which can be executed remotely on the slaves in parallel. An example how one can do this is shown in Chapter 4, where the implementation of matrix multiplication is shown. A major disadvantage is that MPI in its current implementations lacks in fault tolerance. This results in a rather instable execution of MPI applications. Moreover, debugging is really difficult as there is no support for it in **Rmpi**.

All in all this package is a good start in creating parallel programs as this can easy be achieved entirely in R.

For further interface functions supplied by the **Rmpi** package, a more detailed description and further examples please consult the package description Yu (2006).

3.4 The rpvm Package

Parallel virtual machine uses the message passing model and makes a collection of computers appear as a single virtual machine (see Section 2.5.3 for details). The package **rpvm** (Li and Rossini. (2007)) provides an interface to low level PVM functions and a few high level parallel functions to R. It uses most of the facilities provided by the PVM system which makes **rpvm** ideal for prototyping parallel statistical applications in R.

Generally, parallel applications can be either written in compiled languages like C or FORTRAN or can be called as R processes. The latter method is used in this thesis and therefore a good selection of **rpvm** functions are explained in this section to show how PVM and R can be used together.

3.4.1 Initialization and Status Queries

At first for using the PVM system *pvmd3* has to be booted. This can be done via the command line using the `pvm` command (see pages 22 and 23 in Geist et al. (1994)) or directly within R after loading the **rpvm** package using `.PVM.start.pvmd()` explained in this section.

Functions for Managing the Virtual Machine

`.PVM.start.pvmd()` boots the *pvmd3* daemon. The currently running R session becomes the master process.

`.PVM.add.hosts(hosts)` takes a vector of hostnames to be added to the current virtual machine. The syntax of the hostnames is similar to the lines of a pvmd hostfile (for details see the man page of *pvmd3*).

`.PVM.del.hosts()` simply deletes the given hosts from the virtual machine configuration.

`.PVM.config()` returns information about the present virtual machine.

`.PVM.exit()` tells the PVM daemon that this process leaves the parallel environment.

`.PVM.halt()` shuts down the entire PVM system and exits the current R session.

When using a job queueing system like the Sun Grid Engine (SGE) to boot the PVM parallel environment the developer is not engaged with setting up and booting the environment anymore (see appendix B on how to do this).

Example 3.4 shows how the configuration of the parallel environment can be obtained. `.PVM.config()` returns the hosts connected to the parallel virtual machine. After that the parallel environment is stopped.

Example 3.4 Query status of PVM

```
> library("rpvm")
> set.seed(1782)
> .PVM.config()
  host.id    name    arch speed
1  262144 node066 LINUX64  1000
2  524288 node020 LINUX64  1000
3  786432 node036 LINUX64  1000
4 1048576 node016 LINUX64  1000

> .PVM.exit()
```

3.4.2 Process Spawning and Communication

The package **rpvm** uses the master-slave paradigm meaning that one process is the master task and the others are slave tasks. **rpvm** provides a routine to spawn R slaves but these slaves cannot be used interactively like the slaves in **Rmpi**. The spawned R slaves source an R script which contains all the necessary function calls to set up communication and carry out the computation and after processing terminate. PVM uses task IDs (`tid`—a positive integer for identifying a task) and tags for communication (see also the fundamentals of message passing in Section 2.5.1).

Process Management Functions

`.PVM.spawnR(slave, ntask = 1, ...)` spawns `ntask` copies of an `slave` R process. `slave` is a character specifying the source file for the R slaves located in the package's demo directory (the default). There are more parameters indicated by the ... (we refer to Li and Rossini. (2007)). The `tids` of the successfully spawned R slaves are returned.

`.PVM.mytid()` returns the `tid` of the calling process.

`.PVM.parent()` returns the `tid` of the parent process that spawned the calling process.

`.PVM.siblings()` returns the `tid` of the processes that were spawned in a single spawn call.

`.PVM.pstats(tids)` returns the status of the PVM process(es) with task ID(s) `tids`.

3.4.3 Built-in High Level Functions

rpvm provides only two high level functions. One of them is a function to get or set values in the virtual machine settings. This is certainly because each new high level functions needs separate source files for the slaves and this is not what developers do intuitively.

High Level Functions

`PVM.rapply(X, FUN = mean, NTASK = 1))` Apply a function `FUN` to the rows of a matrix `X` in parallel using `NTASK` tasks.

`PVM.options(option, value)` Get or set values of libpvm options (for details see Li and Rossini. (2007) and Geist et al. (1994)).

Example 3.5 shows how the rows of a matrix `X` can be summed up in parallel via `PVM.rapply()`.

Example 3.5 Using PVM.rapply

```
> n <- 8
> X <- matrix(rnorm(n * n), nrow = n)
> round(X, 3)
```

```
       [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]
[1,] -0.200 -0.183  0.560  1.286  0.468  0.502  0.874 -0.778
[2,] -1.371  0.484 -0.498  1.788  0.534 -0.566  0.152 -1.307
[3,]  1.041  0.484  0.399  0.580  0.586 -0.660  1.833 -1.405
[4,] -1.117 -0.893  0.408 -1.612  0.486  0.644  0.422 -1.639
[5,]  1.397 -0.237 -1.287 -0.122 -1.076  0.225 -0.047  0.020
[6,] -0.046  0.537 -1.287 -0.089  0.564  2.671 -0.715 -0.901
[7,]  1.085  0.706 -0.034  0.929  0.057 -2.402 -1.233  1.135
[8,]  0.605 -0.076 -0.554  1.385 -0.436  0.249  0.338  1.369
```

```
> PVM.rapply(X, sum, 3)
```

```
Try to spawn tasks...
Work sent to  524289
Work sent to  786433
Work sent to  1048577
[1]  2.5291297 -0.7840539  2.8572559 -3.3002539 -1.1275728
[6]  0.7329710  0.2431164  2.8806456
```

Before we explain how `PVM.rapply` works the following **rpvm** functions have to be explained:

Other Important functions

`.PVM.initsend()` clears the default send buffer and prepares it for packing a new message.

`.PVM.pkstr(data = "")` and `.PVM.pkint(data = 0, stride = 1)` are low level correspondents of the PVM packing routines (see Geist et al. (1994) for more information on packing data).

`.PVM.pkdblmat(data)` packs a double matrix including the dimension information. There are more packing routines available. They are explained in Li and Rossini. (2007).

`.PVM.send(tid, msgtag)` sends the message stored in the active buffer to the PVM process identified by `tid`. The content is labeled by the identifier `msgtag`.

`.PVM.recv(tid = -1, msgtag = -1)` blocks the process until a message with label `msgtag` has arrived from `tid`. -1 means any. The receive buffer is cleared and the received message is placed there instead.

`.PVM.upkstr()`, `.PVM.upkint()`, `.PVM.upkdblvec()` and others are the corresponding unpack functions to the pack functions explained before.

`.PVM.gather(x, count = length(x), msgtag, group, rootginst = 0)` gathers data distributed on the nodes (`x`) to a specific process (mostly the root) into a single array. It performs a send of messages from each member of a group of processes. A specific process (the root) accumulates this messages into a single vector.

`.PVM.scatter(x, count, msgtag, group, rootqinst = 0)` sends to each member of a group a partition of a vector `x` from a specified member of the group (mostly the root) where `count` is an integer specifying the number of elements to be sent to each member.

Let us now examine the function to see how parallel programs using **rpvm** can be written.

Example 3.6 PVM.rapply master routine

```
PVM.rapply <- function (X, FUN = mean, NTASK = 1)
{
    WORKTAG <- 22
    RESULTAG <- 33
    if (!is.matrix(X)) {
        stop("X must be a matrix!")
    }
    if (NTASK == 0) {
        return(apply(X, 1, FUN))
    }
    end <- nrow(X)
    chunk <- end%/%NTASK + 1
    start <- 1
    mytid <- .PVM.mytid()
    children <- .PVM.spawnR(ntask = NTASK, slave = "slapply")
    if (all(children < 0)) {
```

```
        cat("Failed to spawn any task: ", children, "\n")
        .PVM.exit()
    }
    else if (any(children < 0)) {
        cat("Failed to spawn some tasks.  Successfully spawned ",
            sum(children > 0), "tasks\n")
        children <- children[children > 0]
    }
    for (id in 1:length(children)) {
        .PVM.initsend()
        range <- c(start, ifelse((start + chunk - 1) > end, end,
            start + chunk - 1))
        work <- X[(range[1]):(range[2]), , drop = FALSE]
        start <- start + chunk
        .PVM.pkstr(deparse(substitute(FUN)))
        .PVM.pkint(id)
        .PVM.pkdblmat(work)
        .PVM.send(children[id], WORKTAG)
        cat("Work sent to ", children[id], "\n")
    }
    partial.results <- list()
    for (child in children) {
        .PVM.recv(-1, RESULTAG)
        order <- .PVM.upkint()
        partial.results[[order]] <- .PVM.upkdblvec()
    }
    .PVM.exit()
    return(unlist(partial.results))
}
```

The corresponding slave R script (slapply.R) looks as follows:

Example 3.7 PVM.rapply slave routine

```
library (rpvm)
WORKTAG <- 22
RESULTAG <- 33
mytid  <- .PVM.mytid ()
myparent  <- .PVM.parent ()
## Receive work from parent (a matrix)
buf <- .PVM.recv (myparent, WORKTAG)
## Function to apply
func  <- .PVM.upkstr ()
```

```
cat ("Function to apply: ", func, "\n")
## Order
order <- .PVM.upkint ()
partial.work <- .PVM.upkdblmat ()
print (partial.work)
## actually work, take the mean of the rows
partial.result <- apply (partial.work, 1, func)
print (partial.result)
## Send result back
.PVM.initsend ()
.PVM.pkint (order)
.PVM.pkdblvec (partial.result)
.PVM.send (myparent, RESULTAG)
## Exit PVM
.PVM.exit ()
## Exit R
q (save="no")
```

Example 3.6 shows the master routine of `PVM.rapply()`. This function takes a matrix, the function which is going to be applied and the number of processors to use as arguments. At first the message tags are specified. These tags are necessary to uniquely identify messages sent in a message passing environment. After input validation `NTASK` child processes are spawned using the `.PVM.spawnR()` command. After initializing the send buffer the partitioned data (packed in the buffer using the `.PVM.pk*` commands) is send to the corresponding child processes represented by their task IDs using `.PVM.send()`. PVM uses these task identifiers (tid) to address pvmds, tasks, and groups of tasks within a virtual machine.

Meanwhile the spawned slave processes (see Example 3.7) have been idle because they wait for input (`.PVM.receive()` is a blocking command). After receiving data from the parent the data gets unpacked. Now the slaves apply the given function to their part of the matrix. Finally another send is initialized to provide the results to the parent process and the slaves are detached from the virtual machine by calling a `.PVM.exit()`.

3.4.4 Conclusion

The `PVM.rapply()` example shown in this section followed the Single Program Multiple Data (SPMD) paradigm. Data is split into different parts which are sent to different processes. I/O is handled solely by a master process. When loading **rpvm** in an R session this session becomes the master

process. Slaves can easily be spawned provided that there are working slave scripts available.

We encountered no problems when using the routines in **rpvm**. This package seem to be rather stable in contrast to **Rmpi**, where for unknown reasons the MPI environment sometimes crashed.

A major disadvantage is that the **rpvm** package only has two higher level function. One of them can be used for calculations. That means when using this package for parallel computing one has to deal with low level message passing but which in turn may provide higher flexibility. New parallel functions can be constructed on the basis of the provided interface. The `PVM.rapply` code can be taken as a template for further routines.

Another disadvantage is the missing support for interactive R slaves. Parallel tasks have to be created on the basis of separate slave source files which are sourced on the creation of the slaves.

For further interface functions supplied by the **rpvm** package, a more detailed description and further examples please consult the package description Li and Rossini. (2007).

3.5 The snow Package

The aim of simple network of workstations (**snow**—Rossini et al. (2003), Tierney et al. (2007)) is to provide a simple parallel computing environment in R. To make a collection of computers to appear as a virtual cluster in R three different message passing environments can be used:

- PVM via R package **rpvm** (see section 3.4)
- MPI via R package **Rmpi** (see section 3.3)
- SOCK via TCP sockets

The details of the mechanism used and how it implemented are hidden from the high level user. As the name suggests it should be simple to use.

After setting up a virtual cluster developing parallel R functions can be achieved via a standardized interface to the computation nodes.

Moreover, when using **snow** one can rely on a good handful of high level functions. This makes it rather easy to use the underlying parallel computational engine. Indeed **snow** uses existing interfaces to R namely **Rmpi** when using MPI (see Section 3.3), **rpvm** when using PVM (see Section 3.4) and a new possibility of message passing namely TCP sockets, which is a rather simple way of achieving communication between nodes (in most applications

this is not the optimal way). What follows is a description of high level functions supplied by the **snow** package.

3.5.1 Initialization

Initializing a **snow** cluster is rather easy if the system is prepared accordingly. When using MPI (achieved through **Rmpi**) a LAM/MPI environment has to be booted prior starting the virtual cluster (see Section 3.3). Is PVM the method of choice the **rpvm** package must be available and an appropriate PVM has to be started (see Section 3.4). For both MPI and PVM the parallel environment can be configured through a grid engine (see Appendix B). TCP sockets can be set up directly using the package. MPI or PVM offer the possibility to query the status of the parallel environment. This can be done using the functions supplied from the corresponding package.

Management Functions

`makecluster(spec, type = getClusterOption("type"))` starts a cluster of type `type` with `spec` numbers of slaves. If the cluster is of connection type SOCK then `spec` must be a character vector containing the hostnames of the slavenodes to join the cluster. The return value is a list containing the cluster specifications. This object is necessary in further function calls.

`stopCluster(cl)` stops a cluster specified in `cl`.

Example 3.8 shows how a virtual cluster can be created using MPI.

Example 3.8 Start/stop cluster in **snow**

```
> library("snow")
> set.seed(1782)
> n <- 8
> cl <- makeCluster(n, type = "MPI")
        8 slaves are spawned successfully. 0 failed.

> stopCluster(cl)
[1] 1
```

3.5.2 Built-in High Level Functions

snow provides a good handful of high-level functions. They can be used as building blocks for further high level routines.

High Level Functions

`clusterEvalQ(cl, expr)` evaluates an R expression `expr` on each cluster node provided by `cl`.

`clusterCall(cl, fun, ...)` calls a function `fun` with arguments ... on each node found in `cl` and returns a list of the results.

`clusterApply(cl, x, fun, ...)` applies a function `fun` with additional arguments ... to a specific part of a vector `x`. The return value is of type list with the same length as of `x`. The length of `x` must not exceed the number of R slaves spawned as each element of the vector is used exactly by one slave. To achieve some sort of load balancing please use the corresponding apply functions below.

`clusterApplyLB(cl, x, fun, ...)` is a load balancing version of `clusterApply()` which applies a function `fun` with additional arguments ... to a specific part of a vector `x` with the difference that the length of `x` can exceed the number of cluster nodes. If a node finished with the computation the next job is placed on the available node. This is repeated until all jobs have completed.

`clusterExport(cl, list)` broadcasts a list of global variables on the master (`list`) to all slaves.

`parApply(cl, x, fun, ...)` is one of the parallel versions of the `apply` functions available in R. We refer to the package documentation (Tierney et al. (2007)) for further details.

`parMM(cl, A,B)` is a simple parallel implementation of matrix multiplication.

Example 3.9 shows the use of `clusterApply()` on a virtual cluster of 8 nodes using MPI as communication layer. Like in Example 3.3 a vector of n random numbers is generated on each of the n slaves and are returned to the master as a list (each list element representing one row). Finally an $n \times n$ matrix is formed and printed. The output of the matrix shows again for each row the same random numbers. This is because of the fact, that each slave has the same seed. This problem is treated more specifically in Chapter 5. For more information about parallel random number generators see the descriptions of the packages **rsprng** and **rlecuyer** in Section 3.7.

Example 3.9 Using high level functions of snow

```
> n <- 8
> cl <- makeCluster(n, type = "MPI")
```

```
        8 slaves are spawned successfully. 0 failed.
```

```
> x <- rep(n, n)
> rows <- clusterApply(cl, x, runif)
> X <- matrix(unlist(rows), ncol = n, byrow = TRUE)
> round(X, 3)
```

```
      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]
[1,] 0.691 0.128 0.188 0.206 0.119 0.171 0.873 0.238
[2,] 0.691 0.128 0.188 0.206 0.119 0.171 0.873 0.238
[3,] 0.691 0.128 0.188 0.206 0.119 0.171 0.873 0.238
[4,] 0.691 0.128 0.188 0.206 0.119 0.171 0.873 0.238
[5,] 0.691 0.128 0.188 0.206 0.119 0.171 0.873 0.238
[6,] 0.691 0.128 0.188 0.206 0.119 0.171 0.873 0.238
[7,] 0.691 0.128 0.188 0.206 0.119 0.171 0.873 0.238
[8,] 0.691 0.128 0.188 0.206 0.119 0.171 0.873 0.238
```

3.5.3 Fault Tolerance

Providing fault tolerance, computational reproducibility and dynamic adjustment of the cluster configuration is required for practical real-world parallel applications according to Sevcikova and Rossini (2004b). Failure detection and recovery, dynamic cluster resizing and the possibility to obtain intermediate results of a parallel computation is implemented in the package **snowFT** (Sevcikova and Rossini (2004a)). A detailed introduction to fault tolerance in statistical simulations can be found in Sevcikova (2004). As we pointed out in Section 2.5.2 and Section 2.5.3 MPI does not provide tools for implementing fault tolerance and therefore package **rpvm** is required for **snowFT**. For more details on the fault tolerant version of **snow** we refer to the package documentation (Sevcikova and Rossini (2004a)).

3.5.4 Conclusion

The routines available in package **snow** are easy to understand and use, provided that there is a corresponding communication environment set up. Generally, the user need not know the underlying parallel infrastructure, she just ports her sequential code so that it uses the functions supplied by **snow**. All in all as the title suggests simple network of workstations is simple to

get started with and is simple with respect to the possibilities of parallel computations.

For further interface functions supplied by the **snow** package, a more detailed description and further examples please consult the package description Tierney et al. (2007).

3.6 paRc—PARallel Computations in R

In the course of this thesis a package called **paRc** (Theussl (2007b)) has been developed with the aim to evaluate performance of parallel applications and to show how interfacing high performance applications written in C can be done using OpenMP (see Section 2.5.4).

The package **paRc** contains interface functions to the OpenMP library and provides high level interface functions to a few C implementations of parallel applications using OpenMP (e.g., matrix multiplication—see Chapter 4). Furthermore, it supplies a benchmark environment for performance evaluation of parallel programs and a framework for pricing options with parallel Monte Carlo simulation (see Chapter 5).

paRc can be obtained from `R-Forge.R-project.org`—the R-project community service. To install this package directly within R call
`install.packages("paRc", repos = "R-Forge.R-project.org")`.

To properly install this package you need either the Intel compiler with version 9.1 or newer (the Linux compiler is free for non-commercial use) or the GNU C compiler with version 4.2 or newer. They are known to support OpenMP.

Examples in this section are produced on a bignode of cluster@WU. Bignodes provide a shared memory platform with up to 4 CPUs. Shared memory platforms are necessary for running parallel OpenMP applications.

3.6.1 OpenMP Interface Functions

The user is provided with a few interface functions to the OpenMP library. They are used to query the internal variables of the compiled parallel environment or to change them.

OpenMP Routines

`omp_get_num_procs()` returns the number of threads available to the program.

`omp_set_num_threads()` sets the number of threads to be used in subsequent parallel executions.

`omp_get_max_threads()` gets the number of threads to be used in subsequent parallel executions.

OpenMP Specific Environment Variables

Moreover, environment variables can affect the runtime behavior of OpenMP programs. These environment variables are (OpenMP Architecture Review Board (2005)):

`OMP_NUM_THREADS` sets the number of threads to use in parallel regions of OpenMP programs.

`OMP_SCHEDULE` sets the runtime schedule type and chunk size.

`OMP_DYNAMIC` defines whether dynamic adjustments of threads should be used in parallel regions.

`OMP_NESTED` enables or disables nested parallelism.

Example 3.10 shows the use of the OpenMP library calls in R. First the number of available processors is queried. Then the number of threads a parallel application may use is set to 2. With the last call the current available CPUs to a parallel program is queried.

Example 3.10 OpenMP function calls using **paRc**

```
> library("paRc")
> omp_get_num_procs()
[1] 4

> omp_set_num_threads(2)
> omp_get_max_threads()
[1] 2
```

3.6.2 High Level OpenMP Functions

paRc provides the following high level OpenMP function:

`omp_matrix_multiplication(X, Y, n_cpu = 1)` multiplies the matrix `X` with matrix `Y` using `n_cpu` numbers of processors.

3.6.3 Benchmark Environment

paRc provides a benchmark environment for measuring the performance of parallel programs. Two main functions exist in this context—one for creating a benchmark object and one for running the benchmark described by the object.

Class 'benchmark'

An S3 object (Chambers and Hastie (1991)) of class '`benchmark`' contains all the necessary information to run a benchmark. The elements in the object are

task is a character string defining the task of the benchmark. Currently, the following tasks are implemented:

- matrix multiplication
- Monte Carlo simulation

data is a list containing the parameters and data to properly run the task.

type defines the parallel programming model used to run the benchmark. Currently, the following types are implemented (not necessarily all of them are available for each task):

- OpenMP—C interface calls provided by **paRc**,
- MPI—implementation in **paRc** using **Rmpi** for communication,
- PVM—implementation in **paRc** using **rpvm** for communication,
- snow-MPI—implementation in **snow** using **Rmpi** for communication,
- and snow-PVM—implementation in **snow** using **rpvm** for communication.

cpu_range contains a vector of integers representing the number of CPUs for th corresponding benchmark run.

is_parallel is a logical `TRUE` or `FALSE` whether the benchmark contains parallel tasks or not.

runs is an integer defining the number of repetitions of the benchmark.

Main Routines

These are the main routines for benchmarking parallel applications:

`create_benchmark(task, data, type, cpu_range, ...)` defines a benchmark object using a specific `task` and the corresponding `data`. The `type` refers to the serial or parallel paradigm to use. The `cpu_range` specifies the range of CPUs to use for this benchmark.

`run_benchmark(x)` takes a benchmark object as argument and carries out the defined benchmark. An object of type '`benchmark_results`' is returned containing the results of the benchmark.

Results

Results of the benchmark are stored in a data frame. It is an object of class '`benchmark_results`' and inherits from class '`data.frame`'. Each row represents a run of the dedicated benchmark and for each run the following data is saved in addition to the data defined in the '`benchmark`' object:

`time_usr`, `time_sys`, `time_ela` contain the measured runtimes (measured with the R function `system.time()`).

`run` is an integer representing the number of the benchmark run when using a specific task with a given number of CPUs and a given programming model.

Example 3.11 runs an OpenMP benchmark which returns an object of class '`benchmark_results`'.

Extractor and Replacement Functions

The following routines are for handling a benchmark object. They extract or replace the values in the benchmark `x`.

- `benchmark_task(x)`
- `benchmark_data(x)`
- `benchmark_type(x)`
- `benchmark_cpu_range(x)`

The following routines supply extra information about the benchmark object or the benchmark environment.

`benchmark_is_parallel` returns `TRUE` if the benchmark contains a parallel function to apply.

`benchmark_tasks` returns the tasks which are possible to run with the benchmark environment.

`benchmark_types` returns the types of available serial or parallel paradigms to run with the benchmark environment.

Generic Functions

Generic functions provide methods for different objects. In **paRc** a generic function for calculating the speedup is provided:

`speedup(x)` is a generic function taking an object as an argument. Currently there are two methods implemented namely `speedup.numeric` and `speedup.benchmark_results`. The methods calculate the speedup as it is presented in Equation 2.1.

S3 Methods

The following S3 methods are provided for the benchmark environment:

`print.benchmark` prints objects of class '`benchmark`'

`plot.benchmark_results` supplies a plot method for comparing benchmark results.

`speedup.default` returns an error message that there is no default method.

`speedup.numeric` returns the speedups calculated from a vector of type '`numeric`'. The reference execution time is the first element in the vector. The return value is a vector of type '`numeric`'.

`speedup.benchmark_results` calculates the speedups from a given object of class '`benchmark_results`' and the results are returned as a vector of type '`numeric`'.

Example 3.11 Running a benchmark using OpenMP

```
> n <- 1000
> max_cpu <- omp_get_num_procs()
> dat <- list()
> dat[[1]] <- dat[[2]] <- n
> dat[[3]] <- function(x) {
```

```
+     runif(x, -5, 5)
+ }
> bm <- create_benchmark(task = "matrix multiplication",
+     data = dat, type = "OpenMP", parallel = TRUE,
+     cpu_range = 1:max_cpu)
> bm
```

```
A parallel benchmark running task: matrix multiplication - OpenMP
```

```
> bmres <- run_benchmark(bm)
> bmres
```

```
                  task   type n_cpu time_usr time_sys
2 matrix multiplication OpenMP     1    7.421    0.056
3 matrix multiplication OpenMP     2    7.268    0.060
4 matrix multiplication OpenMP     3    7.666    0.056
5 matrix multiplication OpenMP     4    7.955    0.059
  time_ela is_parallel run
2    7.494        TRUE   1
3    3.697        TRUE   1
4    2.697        TRUE   1
5    2.110        TRUE   1
```

```
> speedup(bmres)
```

```
  OpenMP   OpenMP   OpenMP   OpenMP
1.000000 2.027049 2.778643 3.551659
```

Running a dedicated benchmark is shown in Example 3.11. The object `bm` contains all the information to carry out the benchmark. The task to run is a matrix multiplication of two 1000×1000 matrices. The parallel paradigm to use is OpenMP using 1 up to the maximum of CPUs available on the machine. The resulting data frame is printed and the speedup is calculated.

3.6.4 Environment for Option Pricing

In package **paRc** a framework for pricing financial derivatives, in particular options, is available. The framework is built around the main function namely Monte Carlo simulation and its parallel derivative.

Class 'option'

An S3 object of class 'option' contains all the necessary information for pricing an option. The list elements are

underlying is a numeric vector containing three elements to describe a stock:

- the return μ,
- the volatility σ,
- and the present value of the stock.

strike_price is a numeric defining the strike price of the option.

maturity is a numeric defining the time until the option expires.

type is a character representing the type of the option. There are two possibilities—either a `"Call"` option, or a `"Put"` option.

kind contains a character representing the class of the option. Only options of type `"European"` can be priced at the time of this writing.

position is a character string marking the position of the investor. This can either be `"long"` or `"short"`.

Main Routines

These are the main routines for pricing an option:

`define_option(underlying, strike_price, maturity, ...)` defines an object of class 'option'. Further arguments represented by the ... are `type` (with default `"Call"`), `class` (default `"European"`) and `position` (default `"long"`).

`Black_Scholes_price(x)` takes an object of class 'option' as argument and returns the analytical solution of the Black Scholes differential equation (the price of the option). This works only for European options (see Section 5.2.2 for details).

`Monte_Carlo_simulation(x, r, path_length, n_paths, ...)` carries out a Monte Carlo simulation pricing an option given by `x`. Further arguments are the risk free yield `r`, the length of a Wiener path `path_length`, the number of simulated paths `n_paths`, the number of simulation runs `n_simulations` (with default `50`) and if antithetic variance reduction should be used (`antithetic = TRUE`).

`mcs_Rmpi(n_cpu = 1, spawnRslaves = FALSE, ...)` is the same like the serial version above (`Monte_Carlo_simulation()`) but uses parallel generation of option prices. Additional arguments have to be provided naming the number of CPUs (`n_cpu`) and if R slaves are to be spawned respectively (`spawnRslaves`).

Extractor and Replacement Functions

The following routines are for handling an object of class '`option`'. They extract or replace the values in the option `x`.

- `maturity(x)`
- `strike_price(x)`
- `underlying(x)`
- `option_type(x)`
- `option_class(x)`
- `position(x)`
- `price_of(x)`

S3 Methods

The following S3 methods are provided for the option pricing environment:

`print.option` defines the print method for class '`option`'.

`plot.option` plots the payoff of the given '`option`'.

Example 3.12 Handling class '`option`'

```
> european <- define_option(c(0.1, 0.4, 100), 100,
+     1/12)
> underlying(european)
 mean    sd value
  0.1   0.4 100.0

> option_type(european)
[1] "Call"

> option_class(european)
```

```
[1] "European"

> strike_price(european)

[1] 100

> maturity(european)

[1] 0.08333333

> underlying(european) <- c(0.2, 0.5, 100)
> underlying(european)

 mean    sd value
  0.2   0.5 100.0

> price_of(european) <- Black_Scholes_price(european,
+     r = 0.045)
> european

A Call option with strike price 100
expiring in 30 days
Call price:  5.93155761968948
```

In Example 3.12 the variable `european` is assigned an object of class '`option`'. It is defined as an European option with strike price 100 and maturity 1/12 (30 days). The underlying has a present value of 100, a return of 0.1 and volatility of 0.4. Invoking the print method returns a short summary of the object. The rest of the example shows the use of different extractor functions and an replacement function. Figure 3.6.4 shows the payoff of the defined option.

3.6.5 Other Functions

To complete the set of functions supplied by package **paRc** the following function has to be explained:

`serial_matrix_multiplication(X,Y)` takes the matrices `X` and `Y` as arguments and performs a serial matrix multiplication. This function calls a C routine providing a non-optimized version of the serial matrix multiplication (see Chapter 4 for more details).

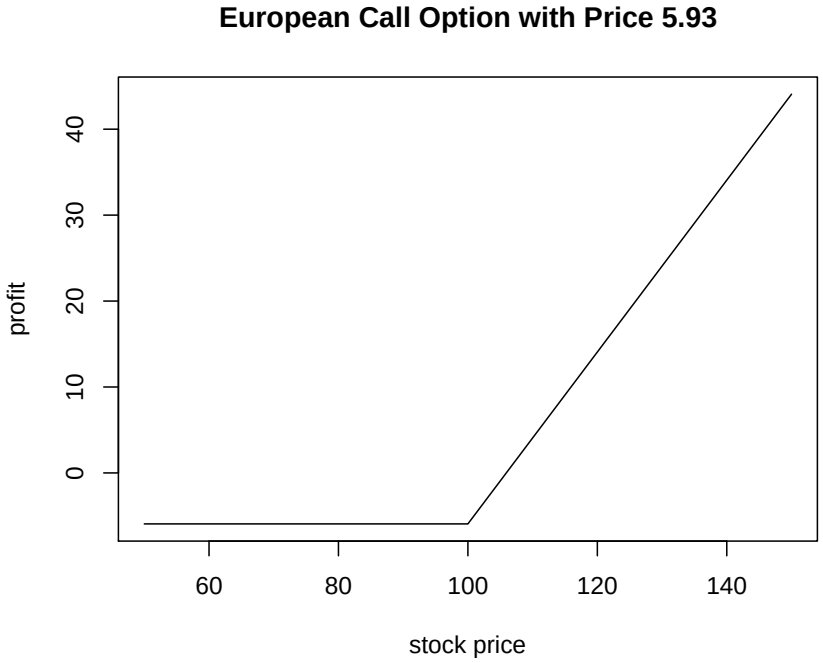

Figure 3.1: Payoff of a European call option with price 5.93

3.7 Other Packages Providing HPC Functionality

There are a few other packages which supply the user with parallel computing functionality or extend other high performance computing packages. In this section a short description of each of these packages is given.

rsprng (Li (2007)) is an R interface to SPRNG (Scalable Parallel Random Number Generators—Mascagni and Srinivasan (2000)). SPRNG is a package for parallel pseudo random number generation with the aim to be easy to use on a variety of architectures, especially in large-scale parallel Monte Carlo applications.

rlecuyer (Sevcikova and Rossini (2004c)) like **rsprng** provides an interface to a parallel pseudo random number generator. **rlecuyer** is the C implementation of the random number generator with multiple independent streams developed by L'Ecuyer et al. (2002).

RScaLAPACK (Samatova et al. (2005) and Yoginath et al. (2005)) uses

the high performance ScaLAPACK library (Dongarra et al. (1997))for linear algebra computations. ScaLAPACK is a library of high performance linear algebra routines which makes use of message passing to run on distributed memory machines. Among other routines it provides functionality to solve systems of linear equations, linear least squares problems, eigenvalue problems and singular value problems. ScaLAPACK is an acronym for Scalable Linear Algebra PACKage or Scalable LAPACK. Samatova et al. (2006) have shown how to use **RScaLAPACK** in biology and climate modelling and analyzed its performance.

papply (Currie (2005)) implements a similar interface to lapply and apply but distributes the processing evenly among the nodes of a cluster. **papply** uses the package **Rmpi** (see Section 3.3) for communication but supports in addition error messages for debugging.

biopara (Lazar and Schoenfeld (2006)) allows users to distribute execution of large problems over multiple machines. It uses R socket connections (TCP) for communication. **biopara** supplies two high level functions, one for distributing work (represented by function and its arguments) among "workers" and the other for doing parallel bootstrapping like the native R function boot().

taskPR (Samatova et al. (2004)) provides a parallel execution environment on the basis of TCP or MPI-2.

3.8 Conclusion

In this chapter we have shown that R offers a lot of possibilities in high performance computing. Extensions which interface the main message passing environments, MPI and PVM, offer the highest flexibility as parallel applications can be based upon low level message passing routines. Especially package **Rmpi** has to be noted as it offers interactive testing of developed parallel functions.

For easy parallelization of existing C code on shared memory machine a new approach has been shown using package **paRc**. Implicit parallel programming can be achieved using the compiler directives offered by OpenMP. A major disadvantage is that parallel functionality has to be implemented low level in C or FORTRAN.

Chapter 4

Matrix Multiplication

4.1 Introduction

If we think of applications in parallel computing matrix multiplication comes to mind. Because of its nature it is a prime example for data parallelism. There has been done a lot of work in this area. E.g., Golub and Van Loan (1996) give a comprehensive introduction to the field of matrix computations.

In this chapter a short introduction to one of the fundamental methods in linear algebra namely matrix multiplication is given. Subsequently an implementation of a serial version of the matrix multiplication algorithm is compared with implementations from high performance linear algebra libraries. The remaining part of this chapter deals with the parallel implementations of matrix multiplication and uses them as a task to benchmark parallel multiplication routines provided from packages presented in Chapter 3. Algorithms from the R package **paRc** are explained in more detail as they have been developed in the course of this thesis. Eventually, results of the comparisons are presented.

4.2 Notation

$\mathbb{R}$ denotes the set of real numbers and $\mathbb{R}^{m \times n}$ the vector space of all m by n real matrices.

$$A \in \mathbb{R}^{m \times n} \Longleftrightarrow A = (a_{ij}) = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} a_{ij} \in \mathbb{R}$$

The lower case letter of the letter which denotes the matrix with subscripts ij

refers to the entry in the matrix.

4.2.1 Column and Row Partitioning

A matrix A can be accessed through its rows as it is a stack of row vectors:

$$A \in \mathbb{R}^{m\times n} \iff A = \begin{pmatrix} a_1^T \\ \vdots \\ a_m^T \end{pmatrix} a_k \in \mathbb{R}^n$$

This is called a *row partition* of A.

If $A \in \mathbb{R}^{m\times n}$ the kth row of A is denoted as $A(k,)$ (according to the row access in R). I.e.,

$$A(k,) = \begin{pmatrix} a_{k1}, & \ldots, & a_{kn} \end{pmatrix}$$

The other alternative is to see a matrix as a collection of column vectors:

$$A \in \mathbb{R}^{m\times n} \iff A = \begin{pmatrix} a_1, & \ldots, & a_n \end{pmatrix} a_k \in \mathbb{R}^m$$

This is called a *column partition* of A.

If $A \in \mathbb{R}^{m\times n}$ the kth column of A can be notated as $A(,k)$ (according to the column access in R). I.e.,

$$A(,k) = \begin{pmatrix} a_{1k} \\ \vdots \\ a_{mk} \end{pmatrix}$$

4.2.2 Block Notation

Block matrices are central in many algorithms. They have become very important in high performance computing because it enables easy distributing of data.

In general an m by n matrix A can be partitioned to obtain

$$A = \begin{pmatrix} A_{11} & \ldots & A_{1q} \\ \vdots & & \vdots \\ A_{p1} & \ldots & A_{pq} \end{pmatrix}$$

A_{ij} is the (i,j) block or sub matrix with dimensions m_i by n_j of A. $\sum_{i=1}^{p} m_i = m$ and $\sum_{j=1}^{q} n_j = n$. We can say that $A = A_{ij}$ is a p by q block matrix.

It can be seen that column and row partitionings are special cases of block matrices.

4.3 Basic Matrix Multiplication Algorithm

Matrix multiplication is one of the fundamental methods in linear algebra. In this section the matrix multiplication problem $C = AB$ is presented using the dot product matrix multiply version. There are others like the saxpy and outer product method which are mathematically equivalent but have different levels of performance as of the different ways they access memory. Nevertheless, we chose the dot product version for illustrating the implementation of the serial version of the matrix multiplication. A good introduction to the other algorithms is given by Golub and Van Loan (1996).

4.3.1 Dot Product Matrix Multiplication

In the usual matrix multiplication procedure the array C is computed through dot products one at a time from left to right and top to bottom order.

$$\mathbb{R}^{m\times r} \times \mathbb{R}^{r\times n} \to \mathbb{R}^{m\times n}$$

$$C = AB \Longleftrightarrow c_{ij} = \sum_{k=1}^{r} a_{ik} b_{kj}$$

Algorithm 1 shows the implementation of the basic dot product matrix multiplication.

Algorithm 1 Basic matrix multiplication algorithm

Require: $A \in \mathbb{R}^{m\times r}$ and $B \in \mathbb{R}^{r\times n}$.
Ensure: $C \in \mathbb{R}^{m\times n}$

```
1: m ← nrow(A)
2: r ← ncol(A)
3: n ← nrow(B)
4: for i = 1 : m do
5:    for j = 1 : n do
6:       for k = 1 : r do
7:          C(i, j) ← C(i, j) + A(i, k)B(k, j)
8:       end for
9:    end for
10: end for
```

4.3.2 C Implementation

It is rather easy to implement Algorithm 1 in C. Example 4.1 shows the serial implementation of the dot product matrix multiplication in package **paRc**. It can be called from R using `serial_matrix_multiplication(A,B)` where `A` and `B` are the matrices to be multiplied.

Example 4.1 Serial matrix multiplication algorithm

```
void Serial_matrix_mult( double *x, int *nrx, int *ncx,
                         double *y, int *nry, int *ncy,
                         double *z) {
  int i, j, k;
  double sum;

  for(i = 0; i < *nrx; i++)
    for(j = 0; j < *ncy; j++){
      sum = 0.0;
      for(k = 0; k < *ncx; k++)
        sum += x[i + k**nrx]*y[k + j**nry];
      z[i + j**nrx] = sum;
    }
}
```

4.3.3 Using Basic Linear Algebra Subprograms

In Section 2.3.2 we pointed out that parallelism can be achieved on the instruction level. Furthermore, we discussed the importance of data locality in memory hierarchies. The Basic Linear Algebra Subprograms (BLAS) standard proposed by Lawson et al. (1979) consists of several basic operations (e.g., vector addition, dot product, etc.) utilizing the underlying architecture optimally. Most attention is paid to the principle of data locality as it will promise the highest performance gain. This can be achieved by moving the block of data once up the memory hierarchy, performing all necessary operations on it and moving the data back to the main memory and proceeding with the next block. The BLAS routines were used in the development of linear algebra libraries for solving a number of standard problems.

We used the following BLAS libraries in benchmarks of the matrix multiplication problem:

refblas is the official implementation from netlib (`http://www.netlib.org`).

Intel MKL is a library designed to offer high performance linear algebra routines on Intel architecture (Intel Corporation (2007c)— available freely for non-commercial use from `http://www.intel.com`).

GotoBLAS is currently the fastest implementation of the Basic Linear Algebra Subprograms (Goto (2007)). This library is threaded, which means that it will use all available number of processors on a target computer. It is freely available for non-commercial purposes from `http://www.tacc.utexas.edu/resources/software/`.

4.3.4 A Comparison of Basic Algorithms with BLAS

To show how the BLAS routines perform on a single machine in comparison to the basic C implementation we ran a benchmark on a bignode of cluster@WU. The complexity of computation is defined as the number of rows and columns a matrix has. We chose to use 1000×1000 and 2500×2500 matrices respectively as grades of complexity (i.e., complexity grade 1000 and complexity grade 2500). The matrix multiplication $C = AB$ is carried out 10 times using two different matrices (A and B). In each run the entries of the matrices are generated using pseudo random numbers of the uniform distribution in the interval -5 to 5. To eliminate fluctuations of the runtime the mean time of these runs is calculated.

Results of Complexity Grade 1000

Table 4.1 shows the results of the matrix multiplication benchmark using the Basic Linear Algebra Subroutines from the libraries mentioned in Section 4.3.3. The entry "native-BLAS" in this table refers to the BLAS library delivered with the R source code. Its aim is to deliver good performance though it has to be of high portability. Furthermore the routines have to produce results with high stability and numerical accuracy of the results has to be maximized. These requirements lead to lower performance compared to the highly specialized libraries.

With libraries trimmed for the corresponding platform a good speedup can be achieved. The Intel MKL library boosts the matrix multiplication with a speedup of over 26 in comparison to the C implementation from Example 4.1. The fastest BLAS library, in this case GotoBLAS, shows a really extraordinary speedup. In fact, nearly doubling the speedup of the MKL. It can really compete with if not beat all parallel algorithms for matrix multiplication shown in this thesis. The drawbacks are that stability of the

results cannot be guaranteed and that this performance gain can only be achieved at the expense of numerical exactness.

	Type	Time	Speedup
1	normal	7.20	1.00
2	native-BLAS	2.56	2.81
3	MKL-BLAS	0.27	26.57
4	goto-BLAS	0.14	52.94

Table 4.1: Comparison of BLAS routines on a bignode with complexity grade 1000

Results of Complexity Grade 2500

Increasing the complexity to 2500×2500 matrices an interesting behavior is shown. Table 4.2 shows that the speedup achieved with the BLAS routines has increased though running the same task on the same machine. The principle of locality is better utilized as more data is been held in higher levels of the memory hierarchies. In fact, the matrices are decomposed in blocks and the probability of using the same data (blocks) again in one of the next calculation steps increases with the complexity of the matrices.

	Type	Time	Speedup
1	normal	141.06	1.00
2	native-BLAS	42.92	3.29
3	MKL-BLAS	3.92	35.95
4	goto-BLAS	1.92	73.36

Table 4.2: Comparison of BLAS routines on a bignode with complexity grade 2500

4.4 Parallel Matrix Multiplication

There are many specialized algorithms for parallel matrix multiplication available. E.g., PUMMA (Parallel Universal Matrix Multiplication Algorithms, see Choi et al. (1993)) or Fox's Algorithm (Fox et al. (1987)) to mention a few. But they are not within the scope of this thesis as we want

to compare the different non-specialized implementations in the R packages with each other.

In this section we first explain in more details the benchmarking process used to produce all the results. The we show how the dot product matrix multiplication can easily be parallelized using OpenMP. Furthermore, the more sophisticated implementations using the MPI and PVM interfaces available in **paRc** are presented. Eventually, these implementations are compared to those available in the package **snow**.

4.4.1 Benchmarking Matrix Multiplication

In general the benchmarks to produce the results in this section are defined and run like in Example 4.2. First, the parameters like the task, the programming model to use in the benchmark or the number of runs are specified. When using the matrix multiplication as a task, data is described as follows: the first two list elements contain the number of rows n and columns m respectively. The third element contains a function for generating the entries of the matrix. It must return a vector of length $n \times m$.

Then the defined benchmark is carried out. For each complexity, each type and each number of CPUs a predefined number of benchmark runs is performed. In each run new entries for the matrices are generated using a predefined random number generator. Then the actual matrix multiplication is carried out. Eventually the results are stored on the hard disk.

Example 4.2 Running a benchmark

```
## definition of benchmark
max_cpu <- mpi.universe.size()
task <- "matrix multiplication"
taskID <- "mm"
paradigm <- "distributed"
types <- c("MPI","snow-MPI", "MPI-wB")
complexity <- c(1000, 2500, 5000)
runs <- 250
## data description
bmdata <- list()
bmdata[[1]] <- bmdata[[2]] <- 1000
bmdata[[3]] <- function(x){
  runif(x,-5,5)
}
## create benchmark object
bm <- create_benchmark(task=task, data=bmdata,
```

```
                      type=types[1], parallel=TRUE,
                      cpu_range=1:max_cpu, runs=runs)
set.seed(1782)
## for each complexity and type run a number of benchmarks
for(n in complexity){
  bmdata[[1]] <- bmdata[[2]] <- n
  benchmark_data(bm) <- bmdata
  for(type in types){
    benchmark_type(bm) <- type
    writeLines(paste("Starting", type ,"benchmark with complexity",
               n, "..."))
    results <- run_benchmark(bm)
    save(results,file=sprintf("%s-%s-%s-%d.Rda", paradigm, taskID,
                              type, n))
  }
}
```

For further examples see Appendix C.

4.4.2 OpenMP

As we mentioned in earlier chapters auto-parallelization using implicit parallel programming is a promising concept. OpenMP is a simple and scalable programming model and with the aim to parallelize existing code without having to completely rewrite (see section 2.5.4). OpenMP directives and routines extend the programming languages FORTRAN and C or C++ respectively to express shared memory parallelism.

OpenMP Directives

OpenMP directives for C/C++ are specified with the `#pragma` preprocessing directive (OpenMP Architecture Review Board (2005)). This means that each OpenMP directive starts with `#pragma omp` followed by directive name and clauses.

Parallel Loop Construct

The parallel loop construct is a combination of the fundamental construct `parallel` which starts parallel execution and a loop construct specifying that the iterations of the loop are distributed across the threads. The syntax of the parallel loop construct `for` is as follows:

- `#pragma omp parallel for` *`optional clauses`*

The *optional clauses* can be any of the clauses accepted by the `parallel` or `for` directives. For example *optional clauses* can be data sharing attribute clauses like the following:

`shared(`*`list`*`)` declares one or more list items to be shared among all the threads,

`private(`*`list`*`)` declares one or more list items to be private to a thread.

The `for` directive places restrictions on the structure of the corresponding loop. For example, the iteration variable is a signed integer, has to be either incremented or decremented in each loop iteration, and is private to each thread. Otherwise threading does not work properly, as loop iterations cannot be decomposed to create parallel threads.

For further information on other directives, constructs, clauses or restrictions to the `for` loop we refer to OpenMP Architecture Review Board (2005).

Implementation

With the above information given we can parallelize the matrix multiplication rather easily.

Algorithm 2 shows how the basic matrix multiplication (see Algorithm 1) can be extended to achieve shared memory parallelism using shared variables.

Algorithm 2 OpenMP matrix multiplication algorithm

Require: $A \in \mathbb{R}^{m \times r}$ and $B \in \mathbb{R}^{r \times n}$.
Ensure: $C \in \mathbb{R}^{m \times n}$

```
1: m ← nrow(A)
2: r ← ncol(A)
3: n ← nrow(B)
4: !$omp parallel for shared(A, B, C, j, k)
5: for i = 1 : m do
6:    for j = 1 : n do
7:       for k = 1 : r do
8:          C(i, j) ← C(i, j) + A(i, k)B(k, j)
9:       end for
10:   end for
11: end for
```

Example 4.3 shows the implementation for Algorithm 2 in C to achieve loop level parallelism. This routine can be called from R using the R function

`omp_matrix_multiplication(A, B, n_cpu)`, where `A` and `B` are the matrices to be multiplied and `n_cpu` is the number of CPUs to be used.

Example 4.3 Parallel matrix multiplication using OpenMP

```
void OMP_matrix_mult( double *x, int *nrx, int *ncx,
                      double *y, int *nry, int *ncy,
                      double *z) {
  int i, j, k;
  double tmp, sum;
  #pragma omp parallel for private(sum) shared(x, y, z, j, k, nrx,
                                               nry, ncy, ncx)
  for(i = 0; i < *nrx; i++)
    for(j = 0; j < *ncy; j++){
      sum = 0.0;
      for(k = 0; k < *ncx; k++)
        sum += x[i + k**nrx]*y[k + j**nry];
      z[i + j**nrx] = sum;
    }
}
```

As OpenMP makes use of shared variables (no message passing) it can only be compared to other paradigms on a shared memory machine. Results are presented in Section 4.5.1.

4.4.3 MPI

Creating parallel applications using message passing is complex as sending and receiving of data has to be done by hand. The developer has to be aware of blocking communication and how to synchronize processes. Furthermore, the sequential program has to be either decomposed with respect to data or with respect to functionality.

For matrix multiplication one wants to apply the same program to different data because a matrix can be decomposed easily to blocks of matrices. Each block is sent to one slave which performs the matrix multiplication on the given data block.

Recalling the block notation from Section 4.2.2 an $m \times n$ matrix can be partitioned to obtain

$$A = \begin{pmatrix} A_1 \\ \vdots \\ A_p \end{pmatrix}$$

A_i is the ith block or sub matrix with dimensions m_i by n of A where $\sum_{i=1}^{p} m_i = m$. We can say that $A = A_i$ is an m_i by n block matrix.

The number of block matrices is given by the number p of available CPUs. With this information given we can write down the matrix multiplication for a specific node:

$$C_i = A_i B$$

C_i represents the local result of the multiplication. Clearly this is not the best way to subdivide matrices in blocks (B has to be sent completely to all nodes) but is a good start to become familiar with data decomposition.

One issue, how many rows are sent to each slave, remains. If the number of rows m of the matrix A can be evenly divided by the number of processors p then m/p rows are sent to each slave. Otherwise the block matrices have to be defined such that $p-1$ slaves receive $\lceil m/p \rceil$ and the remaining processor will receive $m - (p-1)\lceil m/p \rceil$ rows.

This decomposition is used for the implementation of matrix multiplication using MPI and PVM in **paRc**.

Implementation

We used the master-slave parallel programming paradigm (crowd computation) to organize our tasks. This means that a master process exists which is responsible for I/O, process spawning, distributing and collection of the data. Only the slaves carry out the computation.

Package **Rmpi** is used to interface the MPI communication layer. In addition to the high level MPI multiplication routine provided by the package **snow** a separate implementation has been developed for package **paRc** to become familiar with message passing in MPI. The algorithms are presented in this section.

Algorithm 3 shows the instructions to accomplish the task for the master process and Algorithm 4 shows the instructions for the slaves. Whereas the master is engaged with subdividing, sending, and collecting data, the slaves receive their partition of matrix A—A_{rank}, where $rank$ is the identifier of the processor on which the slave runs on. After calculating the partial result C_{rank}, the master gathers them into the resulting matrix C.

Examples 4.4 and 4.5 show the R implementation of Algorithm 3 and Algorithm 4 respectively. After validating the input `n_cpu` number of R slaves are spawned. The master calculates and broadcasts all the necessary data to the slaves. Then the slaves perform the actual computations and return the resulting local matrices to the master. The local solutions are combined to a single matrix which is the solution of the parallel matrix multiplication.

Algorithm 3 MPI matrix multiplication algorithm—master

Require: $A \in \mathbb{R}^{m \times r}$, $B \in \mathbb{R}^{r \times n}$ and p.
Ensure: $C \in \mathbb{R}^{m \times n}$
1: $m \leftarrow \mathbf{nrow}(A)$
2: $n_{slave} \leftarrow \lceil m/p \rceil$
3: $n_{last} \leftarrow m - (p-1)n_{slave}$
4: decompose A to A_i such that $A_1 ... A_{p-1} \in \mathbb{R}^{n_{slave} \times r}$ and $A_p \in \mathbb{R}^{n_{last} \times r}$
5: spawn p slave processes
6: **for** $i = 1 : p$ **do**
7: send A_i, B to process i; Start multiplication on process i
8: **end for**
9: **for** $i = 1 : p$ **do**
10: receive local result C_i from slaves
11: **end for**
12: combine C_i to C

Algorithm 4 MPI matrix multiplication algorithm—slave

Require: $A_{rank} \in \mathbb{R}^{n_{rank} \times r}$, $B \in \mathbb{R}^{r \times n}$, p
Ensure: $C_{rank} \in \mathbb{R}^{n_{rank} \times n}$
1: $C_{rank} \leftarrow A_{rank} B$
2: send local result C_{rank} to master

Example 4.4 MPI master routine

```
mm.Rmpi <- function(X, Y, n_cpu = 1, spawnRslaves=FALSE) {
  dx <- dim(X) ## dimensions of matrix A
  dy <- dim(Y) ## dimensions of matrix B
  ## Input validation
  matrix_mult_validate(X, Y, dx, dy)
  if( n_cpu == 1 )
    return(X%*%Y)
  ## spawn R slaves?
  if(spawnRslaves)
    mpi.spawn.Rslaves(nslaves = n_cpu)
  ## broadcast data and functions necessary on slaves
  mpi.bcast.Robj2slave(Y)
  mpi.bcast.Robj2slave(X)
  mpi.bcast.Robj2slave(n_cpu)
  ## how many rows on slaves
  nrows_on_slaves <- ceiling(dx[1]/n_cpu)
  nrows_on_last <- dx[1] - (n_cpu - 1)*nrows_on_slaves
  ## broadcast number of rows to multiply and slave foo
  mpi.bcast.Robj2slave(nrows_on_slaves)
  mpi.bcast.Robj2slave(nrows_on_last)
  mpi.bcast.Robj2slave(mm.Rmpi.slave)
  ## start partial matrix multiplication on slaves
  mpi.bcast.cmd(mm.Rmpi.slave())
  ## gather partial results from slaves
  local_mm <- NULL
  mm <- mpi.gather.Robj(local_mm, root=0, comm=1)
  out <- NULL
  ## Rmpi returns a list when the vectors have different length
  for(i in 1:n_cpu)
    out <- rbind(out,mm[[i+1]])
  if(spawnRslaves)
    mpi.close.Rslaves()
  out
}
```

Example 4.5 MPI slave routine

```
mm.Rmpi.slave <- function(){
  commrank <- mpi.comm.rank() -1
  if(commrank==(n_cpu - 1))
```

```
    local_mm <- X[(nrows_on_slaves*commrank + 1):(nrows_on_slaves*
                  commrank + nrows_on_last),]%*%Y
  else
    local_mm <- X[(nrows_on_slaves*commrank + 1):(nrows_on_slaves*
                  commrank + nrows_on_slaves),]%*%Y
  mpi.gather.Robj(local_mm,root=0,comm=1)
}
```

4.4.4 PVM

PVM in R can be used with the package **rpvm**. In addition to the high level PVM multiplication routine provided by the package **snow** a separate implementation has been developed for package **paRc** to become familiar with message passing in PVM, which is slightly different from MPI.

Like the MPI implementation above the matrix multiplication is organized according to the master-slave programming model. The difference to MPI is the lack of its ability to remotely execute commands. Therefore when spawning R processes the complete source has to be available to the slave on creation (i.e., from a file). The algorithms for matrix multiplication are the same as shown in Section 4.4.3.

The PVM equivalents to the master and slave code of MPI are shown in Examples 4.6 and 4.7. After validating the input `n_cpu` number of R slaves are spawned. Immediately after invoking this command the R slaves source the code from a file `"mm_slave.R"` (shown in Example 4.7). The master slave calculates the necessary data and stores them in a buffer. This buffer is send to all of the slaves. Then the slaves perform the actual computations (after receiving and unpacking the data of the buffer) and return the resulting local matrices to the master. The local solutions are combined to a single matrix which is the solution of the parallel matrix multiplication.

Example 4.6 PVM master routine

```
mm.rpvm <- function(X, Y, n_cpu = 1) {
  dx <- dim(X) ## dimensions of matrix X
  dy <- dim(Y) ## dimensions of matrix Y
  ## Input validation
  matrix_mult_validate(X,Y,dx,dy)
  ## Tags for message sending
  WORKTAG <- 17
  RESULTAG <- 82
  if(n_cpu == 1)
```

```
    return(X%*%Y)
  mytid <- .PVM.mytid()
  children <- .PVM.spawnR(ntask = n_cpu, slave = "mm_slave.R")
  if (all(children < 0)) {
    cat("Failed to spawn any task: ", children, "\n")
    .PVM.exit()
  }
  else if (any(children < 0)) {
    cat("Failed to spawn some tasks.  Successfully spawned ",
        sum(children > 0), "tasks\n")
    children <- children[children > 0]
  }
  nrows_on_slaves <- ceiling(dx[1]/n_cpu)
  nrows_on_last <- dx[1] - (n_cpu - 1)*nrows_on_slaves
  ## distribute data
  for (id in 1:length(children)) {
    .PVM.initsend()
    .PVM.pkint(id)
    .PVM.pkint(n_cpu)
    .PVM.pkint(nrows_on_slaves)
    .PVM.pkint(nrows_on_last)
    .PVM.pkdblmat(X)
    .PVM.pkdblmat(Y)
    .PVM.send(children[id], WORKTAG)
   }
  ## receive partial results
  partial_results <- list()
  for (child in children) {
    .PVM.recv(-1, RESULTAG)
    rank <- .PVM.upkint()
    partial_results[[rank]] <- .PVM.upkdblmat()
  }
  .PVM.exit()
  ## return in matrix form
  out <- NULL
  for(i in 1:n_cpu)
    out <- rbind(out,partial_results[[i]])
  out
}
```

Eventually, the corresponding slave routine which actually carries out the whole matrix multiplication looks as follows.

Example 4.7 PVM slave routine

```
library("rpvm")
WORKTAG <- 17
RESULTAG <- 82
myparent  <- .PVM.parent ()
## Receive work from parent (a matrix)
buf <- .PVM.recv (myparent, WORKTAG)
rank <- .PVM.upkint() - 1
n_cpu <- .PVM.upkint()
nrows_on_slaves <- .PVM.upkint()
nrows_on_last <- .PVM.upkint()
X <- .PVM.upkdblmat()
Y <- .PVM.upkdblmat()
if(rank==(n_cpu - 1))
  local_mm <- X[(nrows_on_slaves*rank + 1):(nrows_on_slaves*rank +
                nrows_on_last),]%*%Y
if(rank<(n_cpu - 1))
  local_mm <- X[(nrows_on_slaves*rank + 1):(nrows_on_slaves*rank +
                nrows_on_slaves),]%*%Y
## Send result back
.PVM.initsend()
.PVM.pkint(rank + 1)
.PVM.pkdblmat(local_mm)
.PVM.send (myparent, RESULTAG)
## Exit PVM
.PVM.exit ()
## Exit R
q (save="no")
```

4.5 Comparison of Parallel Routines

In this section the results of the comparison of parallel programming models are shown. They are compared on a shared memory machine using up to 4 cores and on a distributed memory platform using up to 20 cluster nodes. On the shared memory machine matrices with dimensions of 1000×1000 (complexity grade 1000) and 2500×2500 (complexity grade 2500) are used and on the cluster nodes matrices with dimensions 2500×2500 and 5000×5000 (complexity grade 5000).

4.5.1 Shared Memory Platform

On our shared memory testbed the number of available CPUs is restricted to four. Benchmarks are run on the bignodes of cluster@WU.

Results of Complexity Grade 1000

Table 4.3 and Figure 4.1 show the resulting execution times of the matrix multiplication with complexity grade 1000 (the dotted line represents the serial reference benchmark). A slight overhead is observed for the OpenMP routine from package **paRc** using one CPU but when increasing the number of CPU this programming model scales better than the other two. This is because OpenMP makes use of shared variables and therefore not as much communication overhead is produced as when using message passing—computation itself does not matter in relation to communication. Nevertheless, a slight overhead produced from the compiler remains. This is what we expected. In this benchmark MPI and PVM show nearly the same speedup.

	#CPU	OpenMP	MPI	PVM
7	1	7.43	7.18	7.20
14	2	3.71	3.97	5.04
21	3	2.70	3.07	4.22
28	4	2.14	2.60	3.97

Table 4.3: Comparison of parallel programming models on a shared memory machine and complexity grade 1000

When using BLAS (as the supplied matrix multiplication routines from **snow** do) the results draw the following picture: Table 4.4 and Figure 4.2 show that all of the programming models scale not so well. The communication overhead is too large for this kind of complexity grade (the BLAS routines deliver results fast). Indeed, when using all computation cores the PVM routine from package **snow** is slower than with fewer cores. This is because the distribution of the data consumes more time with respect to the gain of distributing the computation.

Results of Complexity Grade 2500

Let the complexity be increased to 2500×2500 matrices. Then, in contrary to 4.5.1 Figure 4.3 shows that the OpenMP routine performs worse than the other two. This is because the communication does not play an important role any more when carrying out this benchmark. The compiler driven

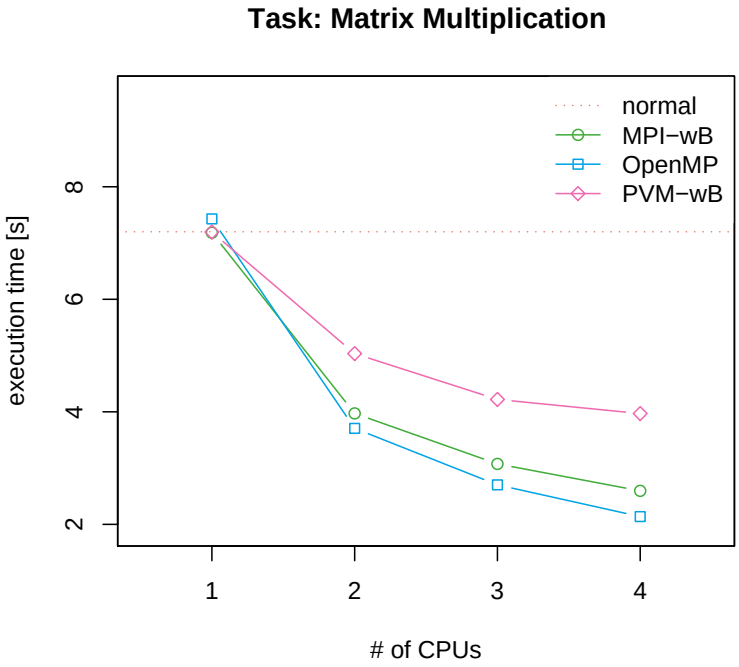

Figure 4.1: Comparison of parallel programming models on a shared memory machine and complexity grade 1000

parallelization produces a slight overhead in contrast to the manual parallelization. The MPI and PVM routines scale nearly in the same way (see also Table 4.5.

The last comparison of parallel programming models on a shared memory machine shows parallel matrix multiplication using BLAS. As we have seen in the corresponding example from Section 4.5.1 the speedup is not expected to be high. Figure 4.4 shows that when increasing the number of cores from one to two a significant speedup can be obtained. Using more than two processors does not improve performance as the lines in the graph become rather flat. Again, the PVM routine from **snow** performs worst. The execution times and speedups are shown in Table 4.6.

4.5.2 Distributed Memory Platform

On a distributed memory platform things change. First more CPUs and more memory are available to the user. Second the processors are connected through a slower interconnection network compared to a shared memory

	#CPU	MPI	snow-MPI	PVM	snow-PVM
3	1	2.55	2.56	2.56	2.56
12	2	1.66	1.81	1.14	1.81
19	3	1.51	1.55	1.18	1.54
26	4	1.38	1.68	1.43	1.49

Table 4.4: Comparison of parallel programming models on a shared memory machine using BLAS and complexity grade 1000

	#CPU	OpenMP	PVM
6	1	142.63	141.12
11	2	70.45	59.93
16	3	49.07	44.23
21	4	40.10	35.43

Table 4.5: Comparison of parallel programming models on a shared memory machine and complexity grade 2500

machine. This has an influence to all communication carried out in the message passing environment. As we have more nodes available we decided to use bigger matrices and therefore start with 2500×2500 and continue with 5000×5000 matrices.

Results of Complexity Grade 2500

Without using BLAS the MPI routine from package **paRc** scales rather well (Figure 4.5). The PVM routine shows good speedup until processor number 6 but then performance gain breaks down. Again, PVM communication seems to be suboptimal implemented.

Figure 4.6 is interesting. Only the MPI routine from **paRc** seems to deliver good results. The PVM and the **snow** routines deliver lower performance with increasing CPU counts after adding the 4th or 5th CPU. Furthermore the **snow** PVM routine produced 2 outliers. It does not seem to deliver a stable performance.

To sum up Table 4.7 shows the best results of the matrix multiplication benchmark for each parallel programming model. It can be seen that BLAS are highly optimized for the corresponding platform and can hardly be beaten. Looking at the message passing models, the MPI routines perform better with respect to the highest speedup which can be achieved than the PVM ones.

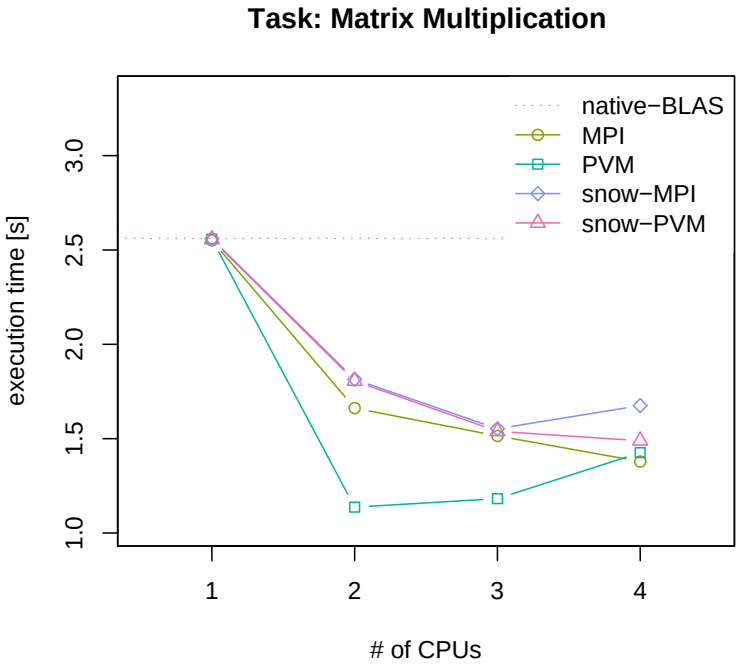

Figure 4.2: Comparison of parallel programming models on a shared memory machine using BLAS and complexity grade 1000

Results of Complexity Grade 5000

The final task in this section is to multiply two 5000×5000 matrices on the cluster nodes. Communication now really matters as one matrix needs around 190 MB of memory when assuming that an entry is a double precision floating point of 8 Byte.

Figure 4.7 shows that the PVM routine has once more problems with large CPU counts. MPI is rather stable. Figure 4.8 shows the routines using the R BLAS library. Again PVM is unstable, the **snow** routines do not scale well and MPI delivers stable and increasing performance.

Finally, Table 4.7 shows the best results of the matrix multiplication benchmark for each parallel programming model with complexity grade 5000. It can be seen that BLAS achieve again the best performance for the corresponding platform (the disadvantages taken into consideration). In fact, comparing the best speedups which can be achieved for each programming model draws the same picture as in Table 4.7: the MPI routines perform better than the PVM routines. The message passing routine from package **paRc** seems to give a better speedup than the others.

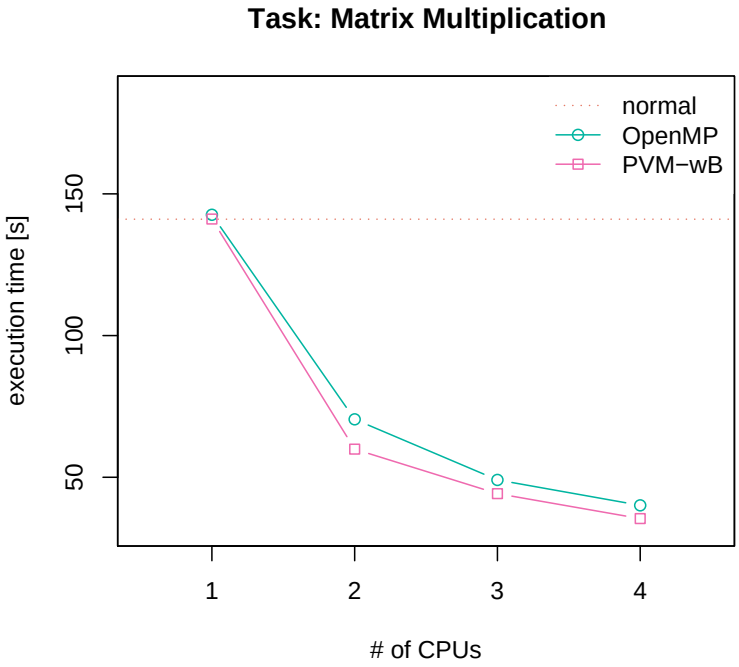

Figure 4.3: Comparison of parallel programming models on a shared memory machine and complexity grade 2500

4.6 Conclusion

In this chapter we have shown how we achieved parallelism in R using OpenMP and using message passing models like MPI and PVM. Furthermore we compared the programming models on a shared and a distributed memory platform.

On the shared memory platform we showed that OpenMP can be easy implemented and that performance is rather good compared to the other parallel programming models (when ignoring the optimized BLAS implementations). Therefore we propose to use this paradigm for parallelization of existing implementations in C as it is possible to incrementally parallelize code. Then, as many commodity computers offer more than one processor nowadays, one can experience higher performance when using R on a shared memory machine.

In contrast, on distributed memory platforms, one has to rely on message passing environments as they are by now the best way of achieving high performance on these machines. It has been shown that MPI delivers better

	#CPU	MPI	PVM	snow-PVM
3	1	42.92	42.94	42.92
10	2	28.40	3.38	28.48
15	3	27.37	3.85	27.28
20	4	27.17	4.58	25.80

Table 4.6: Comparison of parallel programming models on a shared memory machine with BLAS and complexity grade 2500

	#CPU	Type	Time	Speedup
6	1	normal	125.91	1.00
14	2	OpenMP	66.12	1.90
5	1	native-BLAS	28.11	4.48
58	9	PVM-wB	24.65	5.11
42	6	snow-PVM	15.03	8.38
47	7	snow-MPI	14.34	8.78
98	16	MPI-wB	11.66	10.80
97	16	MPI	9.24	13.62
15	2	PVM	8.04	15.67
2	1	MKL-BLAS	3.66	34.43
1	1	goto-BLAS	1.94	64.93

Table 4.7: Comparison of parallel programming models on a cluster using 20 nodes and complexity grade 2500

results in comparison to PVM when used in combination with R. Moreover, it is better to implement this type of parallel applications per hand as high level functions, provided for example by package **snow**, perform not so well. This is because they rely on generalized functions which is suboptimal in the field of high performance computing.

MPI provided by package **Rmpi** seems to be a good choice when building large-scale parallel applications in R. Large data sets can be transmitted via MPI efficiently and processes can be handled interactively.

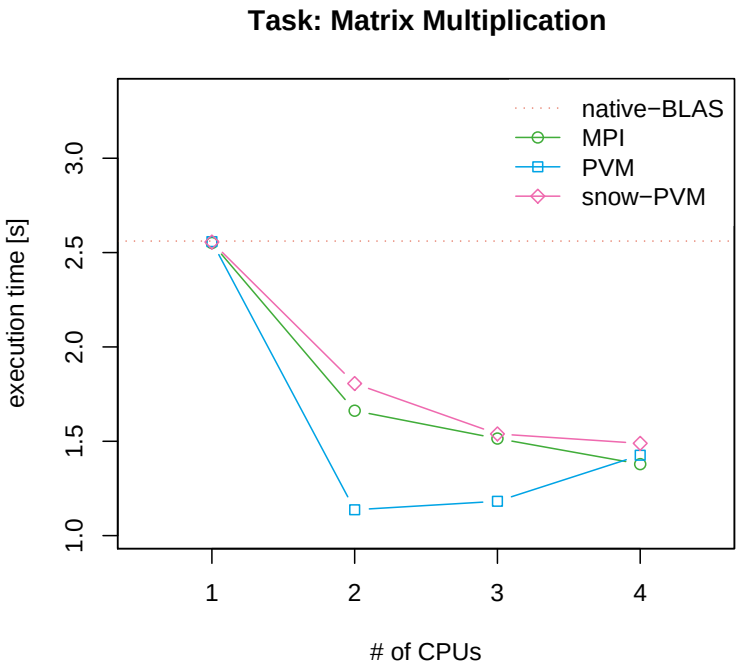

Figure 4.4: Comparison of parallel programming models on a shared memory machine with BLAS and complexity grade 2500

	#CPU	Type	Time	Speedup
6	1	normal	1252.51	1.00
14	2	OpenMP	661.45	1.89
5	1	native-BLAS	222.61	5.63
40	6	PVM-wB	201.79	6.21
59	9	snow-MPI	81.56	15.36
66	10	snow-PVM	72.77	17.21
122	20	MPI-wB	64.10	19.54
121	20	MPI	51.78	24.19
15	2	PVM	35.29	35.49
2	1	MKL-BLAS	28.82	43.45
1	1	goto-BLAS	15.00	83.51

Table 4.8: Comparison of parallel programming models on a cluster using 20 nodes and complexity grade 5000

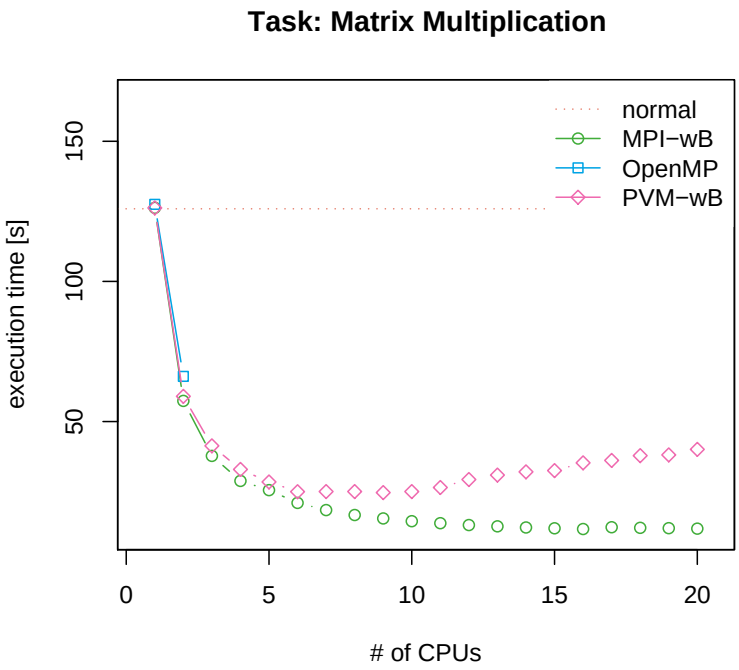

Figure 4.5: Comparison of parallel programming models on a cluster using 20 nodes and complexity grade 2500

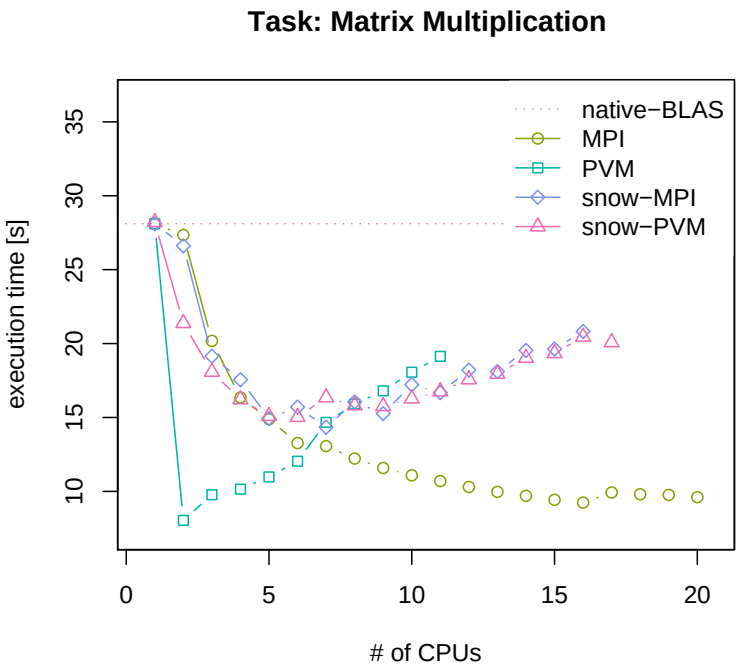

Figure 4.6: Comparison of parallel programming models on a cluster using 20 nodes using BLAS and complexity grade 2500

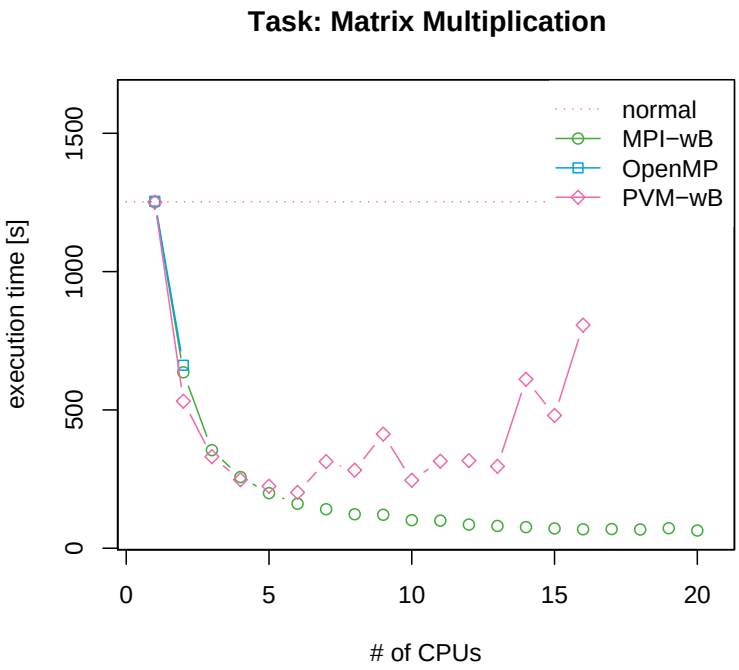

Figure 4.7: Comparison of parallel programming models on a cluster using 20 nodes and complexity grade 5000

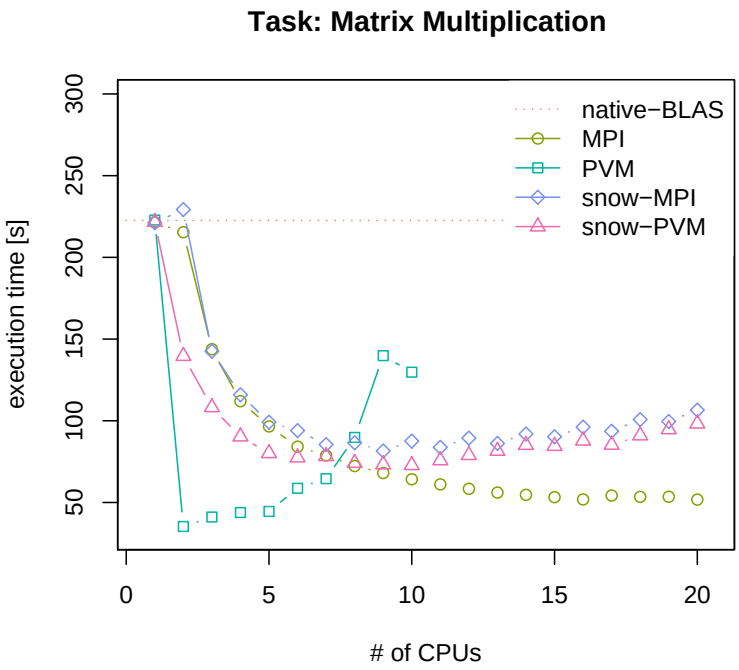

Figure 4.8: Comparison of parallel programming paradigms on a cluster using 20 nodes with BLAS and complexity grade 5000

Chapter 5

Option Pricing using Parallel Monte Carlo Simulation

5.1 Introduction

The search for competitive advantages and especially in economics and in financial markets is in man's nature. The biggest advantage a trader in financial markets can have is to know something before all other market participants. On the stock exchange quickness and an advantage in time are the key factors to be successful. On today's financial markets more and more trades are made by software and their algorithms (this is called algorithmic trading) or even artificial intelligence (Kelly (2007)). A human trader could process five to ten complex transactions per minute software in contrast could process several thousand transactions per second and parallelism preconditioned, look after several market places simultaneously.

The performance of the electronic exchange system is of great importance nowadays. Every millisecond counts and therefore financial firms keep their trading software as close to the trading system as possible (MacSweeney (2007)). This is because investors want to keep the time between the input of a trade until its confirmation (this is called round tripping) at a minimum. Recently the Gruppe Deutsche Börse upgraded their trading system Xetra to offer more bandwidth with lower latency (Gruppe Deutsche Börse (2007b), Gruppe Deutsche Börse (2007a)). These developments show the importance of high performance computing in this sector.

In this chapter we investigate how high performance computing can be applied in finance. This is done as a case study in which a European call option is priced using parallel Monte Carlo simulation and the outcome is compared to the analytical solution.

This chapter is organized as follows: First, a short overview of the theory of option pricing is given, starting with a definition of derivatives and continuing with the explanation of the Black-Scholes model. Then the theory of Monte Carlo simulation of option prices is summarized. Before showing the results of the parallel Monte Carlo simulation the implementation is described.

5.2 Option Pricing Theory

5.2.1 Derivatives

Derivatives are important in today's financial markets. Futures and options have been increasingly traded since the last 20 years. Now, there are many different types of derivatives.

A derivative is a financial instrument whose value depends on the value of an other variable (or the values of other variables). This variable is called *underlying*.

Derivatives are traded on exchange-traded markets (people trade standardized contracts defined by the exchange) or over-the-counter markets (trades are done in a computer linked network or over the phone—no physical contact).

There are two main types of derivatives, namely forwards (or futures) and options.

A forward is a contract in which one party buys (long position) or sells (short position) an asset at a certain time in the future for a certain price. **Futures** in contrast are standardized and therefore are normally traded on an exchange.

A call option is a contract which gives the holder the right to buy an asset at a certain time for a certain price (strike price).

A put option is a contract which gives the holder the right to sell an asset at a certain time for a certain price.

Forward prices (or future prices) can be determined in a simple way and therefore it is not of computational interest for this chapter.

Options on the other hand are more sophisticated. There exist many variants of options on financial markets. European and American options are the most common of them.

European options are derivatives with the right to buy the underlying S at a certain time T (expiration date or maturity) paying a previously defined price X (call option). If one sells this right it is called a put option. C_t (P_t) denotes the value of the call option (put option) at a specific time t. S_t denotes the value of the underlying at a specific time t. The price of a call option is defined as follows:

$$C_T = \left\{ \begin{array}{ll} 0 & \text{if } S_T \leq X \\ S_T - X & \text{if } S_T > X, \end{array} \right\} \text{ or } \max(S_T - X, 0), \tag{5.1}$$

and for the put option:

$$P_T = \left\{ \begin{array}{ll} 0 & \text{if } X \leq S_T \\ X - S_T & \text{if } X > S_T \end{array} \right\} \text{ or } \max(X - S_T, 0). \tag{5.2}$$

Figure 5.2.1 shows the payoff functions of a call and a put option and the position of the investor (left: long position, right: short position).

American options are heavily traded on exchanges. The difference to its European counterpart is the fact, that it can be exercised at any time t within the maturity $0 \ldots T$. The option expires if it is not exercised. The price of an American call option is

$$C_t = \left\{ \begin{array}{ll} 0 & \text{if } S_t \leq X \\ S_t - X & \text{if } S_t > X, \end{array} \right\} \text{ or } \max(S_t - X, 0), \tag{5.3}$$

and for the put option:

$$P_t = \left\{ \begin{array}{ll} 0 & \text{if } X \leq S_t \\ X - S_t & \text{if } X > S_t \end{array} \right\} \text{ or } \max(X - S_t, 0). \tag{5.4}$$

In this chapter the pricing of European options is shown as we can obtain the prices analytically (using the Black-Scholes-Merton model) and compare it to the prices estimated with Monte Carlo simulation.

5.2.2 Black-Scholes Model

In the 1970s Black and Scholes (1973) and Merton (1973) made a major breakthrough in the pricing of options. This soon became known as the Black-Scholes model. They were awarded with the Nobel prize for economics in 1997 for their work (Fischer Black died in 1995, otherwise he would have received this prize too).

In the model developed by Black, Scholes and Merton, price changes are described mathematically by Markov processes. The Brownian motion (or Wiener process) is one of the forms of Markov processes used as a model for stock price movements (this goes back to Bachelier (1900)).

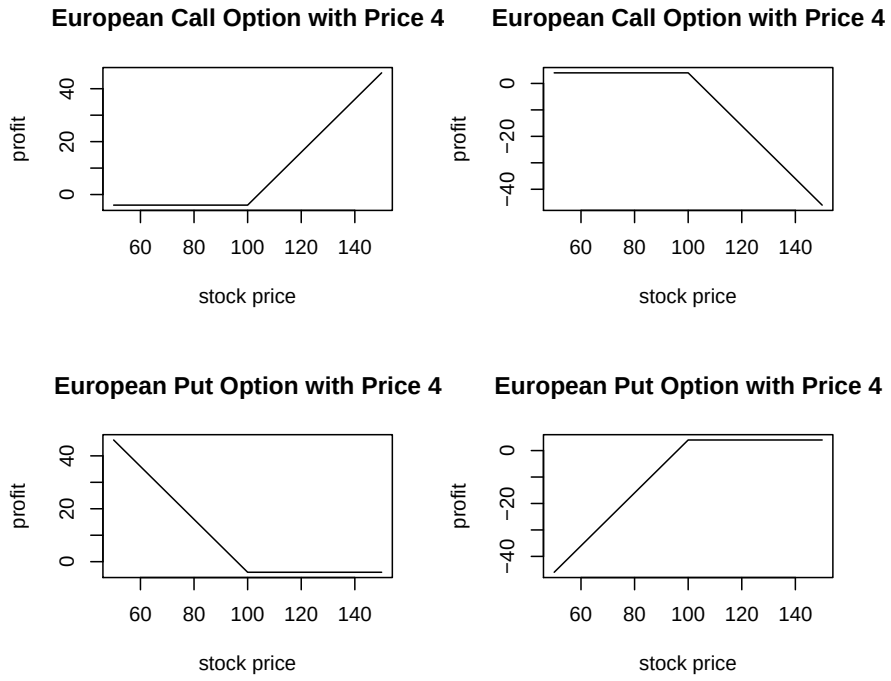

Figure 5.1: Payoffs of a European put and call option depending on the position of the investor

Brownian Motion or Wiener Process

A process $(W_t)_{0 \leq t < \infty}$ is a Brownian motion or a Wiener process if

1. $W_0 = 0$,
2. the paths are continuous,
3. all increments $W_{t_1}, W_{t_2} - W_{t_1}, \ldots, W_{t_n} - W_{t_{n-1}}$ are independent and normal distributed for all $0 < t_1 < t_2 < \cdots < t_n$

The Wiener process is a fundamental part of deriving option pricing formulas. An example of a Wiener path can be seen in Figure 5.2.2.

Stock Price Behavior

The value S of an underlying security follows the most widely used model for stock price behavior, the stochastic process

$$dS = \mu S dt + \sigma S dW \tag{5.5}$$

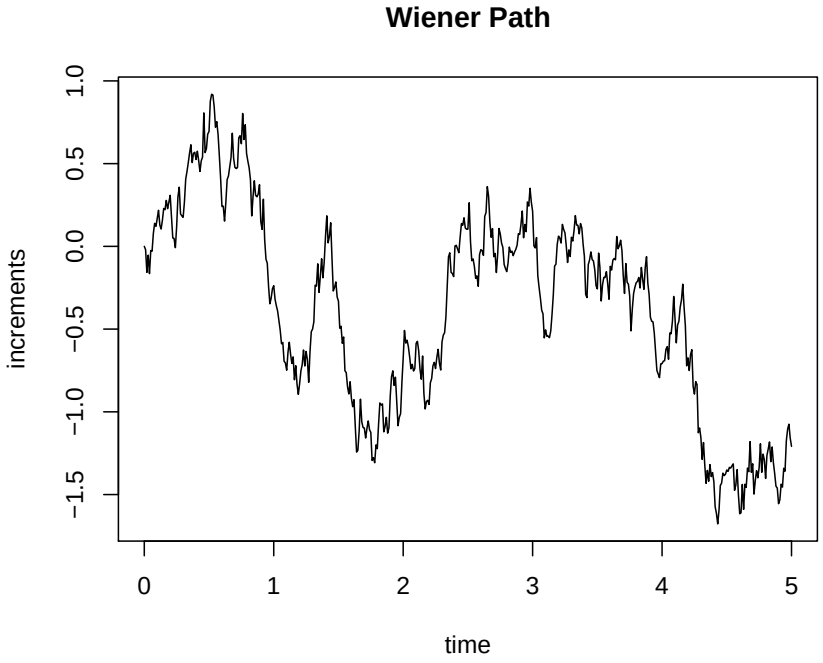

Figure 5.2: Path of a Wiener process

or

$$\frac{dS}{S} = \mu dt + \sigma dW$$

where μ is the drift rate (or expected return) and σ is the volatility. W_t is a standard Brownian motion and dt is a time increment. At $t = 0$ the value $S_{(0)}$ is S_0. If a volatility of zero is assumed the stock price would grow at a continuously compounded rate of μ as

$$\frac{dS}{S} = \mu dt$$

and integrated between 0 and T is

$$S_T = S_0 e^{\mu T}.$$

Black-Scholes Differential Equation

Generally, the price of a derivative is a function of the stochastic variable underlying the derivative (e.g., the stock S) and time t. In this case the stock price process is assumed to be like in Equation 5.5. If we suppose that

V is the value (price) of an option or other derivative, the variable V must be some function of S and t. After applying Itô's Lemma (Itô (1951)) we get

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0. \tag{5.6}$$

Equation 5.6 is known as the Black-Scholes-Merton differential equation. Solving this equation depends on the boundary conditions that are used for a particular derivative. In the case of a European option the boundary condition equals at $t = T$ to the payoff $V(T, S)$. For a call option the value is derived as follows

$$V(T, S) = \max(S(T) - X, 0)$$

and for a put option

$$V(T, S) = \max(X - S(T), 0).$$

Black-Scholes Pricing Formulas

Solving this differential equation (Equation 5.6) subject to the boundary conditions yields to the Black-Scholes formulas for the prices at time zero of European options on a non-dividend paying stock. The price of a call option c is (Hull (2003)):

$$c = S_0 N(d_1) - Xe^{-rT} N(d_2) \tag{5.7}$$

and the price of the corresponding put option p is

$$p = Xe^{-rT} N(-d_2) - S_0 N(-d_1) \tag{5.8}$$

where

$$d_1 = \frac{\log(\frac{S_0}{X}) + (r + \frac{\sigma^2}{2})T}{\sigma\sqrt{T}}$$

$$d_2 = \frac{\log(\frac{S_0}{X}) + (r - \frac{\sigma^2}{2})T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}$$

The function $N(x)$ denotes the cumulative probability distribution function for a standard normal distribution. It returns the probability that a variable sampled from the standard normal distribution is less than x.

Risk Neutral Valuation

The assumption of a risk neutral world is one of the most important tools for the analysis of derivatives. This assumption can be derived from the Black-Scholes differential equation as it does not involve any variable which is affected by the risk preferences of investors (Hull (2003)).

In a world where investors are risk neutral, the expected return on all securities is the risk free rate of interest r. Therefore, the present value of any cash flow can be obtained by discounting its expected value at the risk free yield.

With this information given a derivative can be valued in a risk neutral world by using the following important assumption: In a risk neutral world the return of the underlying asset (μ) is equal to the risk free interest rate (r).

Implementation in Package paRc

The Black-Scholes pricing formula is implemented in package **paRc**. If a European option is available one may call the function `Black_Scholes_price(x, r)` to retrieve the exact price where `x` is an object of class '`option`' and `r` is the risk free yield.

Example 5.1 shows how the price of an European call option with strike price 100 and maturity 30 days (1/12 years) can be calculated using the function `Black_Scholes_price()`. The underlying is a stock with a current value of 100, a μ of 0.1, and volatility σ of 0.4. The risk free yield is assumed to be 4.5%.

Example 5.1 Calculating the Black-Scholes price of an European option

```
> european <- define_option(c(0.1, 0.4, 100), 100, 1/12)
> bsprice <- Black_Scholes_price(european, 0.1)
> bsprice

   value
5.011784
```

The European options can be solved analytically as we saw in this section. Other derivatives can only be priced by numerical solutions. In the remainder of this chapter the use of Monte Carlo simulation is shown. Again we use European options so that we can compare the analytical exact solution with the numerical approximation.

5.2.3 Monte Carlo Evaluation of Option Prices

Monte Carlo simulation is based upon Monte Carlo integration which in turn is based upon the law of large numbers: Let $X_1, X_2, \ldots, X_n$ be independent and identically distributed with a finite expectation μ, then the sequence of arithmetic means $(\overline{X}_n)$ converge almost surely to μ.

Let $f(x)$ be the density of X with the expectation

$$\mathbb{E}[g(X)] = \int g(x)f(x)dx$$

.

With the possibility to generate independent random numbers X_i with the same density of X, the expectation can be estimated as follows:

$$\mathbb{E}[g(X)] = \frac{g(X_1) + \cdots + g(X_n)}{n}$$

The integral

$$\mu = \int g(x)dx$$

can be approximated through

$$\int g(x)dx = \int \frac{g(x)}{f(x)} f(x)dx = \mathbb{E}[\frac{g(X)}{f(X)}]$$

where f is a density with $\{x|g(x) \neq 0\} \subseteq \{x|f(x) > 0\}$.

Then $\int g(x)dx$ can be estimated as follows:

$$\widehat{\mu}_n = \frac{1}{n}\left(\frac{g(X_1)}{f(X_1)} + \cdots + \frac{g(X_n)}{f(X_n)}\right)$$

The accuracy of the estimate can be obtained using the central limit theorem. Let $X_1, X_2, \ldots, X_n$ be independent and identically distributed with expectation μ and variance σ^2, then

$$\frac{\overline{X}_n - \mu}{\sqrt{\frac{\sigma^2}{n}}} \sim N(0,1).$$

$\frac{\sigma}{\sqrt{n}}$ is called standard error. This implies that when increasing n the quality (or accuracy) of the estimate increases.

Simulating Paths of Stocks

The process of the underlying market value of a derivative is shown in Equation 5.5. To simulate the path followed by S the time to maturity of the option has to be subdivided into N short intervals of length δt and approximate Equation 5.5 to

$$S(t+\delta t) - S(t) = \mu S(t)\delta t + \sigma S(t)\epsilon\sqrt{\delta t}$$

where $S(t)$ denotes the value of S at time t, and ϵ is a random sample from a standard normal distribution. Now, for each δt all values of S can be calculated starting from the initial value S_0. One simulation trial involves constructing a complete path for S using N random samples from a normal distribution.

Usually it is more accurate to simulate $\ln S$ rather than S. This leads to

$$S(t+\delta t) = S(t)e^{(\widehat{\mu}-\frac{\sigma^2}{2})\delta t+\sigma\epsilon\sqrt{\delta t}}$$

Here the logarithm of the stock price is thus normally distributed, and the stock price itself has a lognormal distribution. In contrast to the equation above this equation is true for all δt.

The major advantage of Monte Carlo simulation is that the payoffs of options can be simulated which depends not only on the final value of S but also on the whole path followed by the underlying variable S. Furthermore, Monte Carlo simulation can be used to value options which depend on more than one market variable. The drawbacks are that Monte Carlo simulation is computationally very time consuming but makes it in turn to a prime example for parallel computing. A further disadvantage is that it cannot easily handle situations where there are early exercise opportunities (e.g., American options).

Variance Reduction Procedures

It is very expensive (in terms of computation time) to calculate the option price with Monte Carlo simulations as a very large number of simulation runs is necessary to estimate the price with reasonable accuracy. But there are a number of variance reduction procedures available that can lead to savings in computation time.

Among these variance reduction procedures there is a technique called **antithetic variable**. When using antithetic variable two values of the derivative is calculated. One in the usual way (say f_1) and the other using the same random samples of the standard normal distribution but with inverse

sign (f_2). Then, the sample value of the derivative is calculated as the mean of f_1 and f_2. Now, if one of these values is above the true value the other tends to be below and the other way round.

Further information about variance reduction techniques and details about other procedures can be found in Glasserman (2004).

5.3 Monte Carlo Simulation

The value of a derivative can be estimated using the following steps (Hull (2003)):

1. Sample a random path for S in a risk neutral world
2. Calculate the payoff from the derivative
3. Repeat steps 1 and 2 to get many sample values of the payoff from the derivative in a risk neutral world.
4. Calculate the mean of the sample payoffs to get an estimate of the expected payoff in a risk neutral world
5. Discount the expected payoff at a risk free rate to get an estimate of the value of the derivative.

The number of trials carried out depends on the accuracy required. If n_{sim} simulations are run the standard error of the estimate μ of the payoff is

$$\frac{\sigma}{\sqrt{n_{sim}}}$$

where σ is the standard deviation of the discounted payoff given by the simulation for the derivative.

A 95% confidence interval for the price f of the derivative is given by

$$\mu - \frac{1.96\sigma}{\sqrt{n_{sim}}} \leq f \leq \mu + \frac{1.96\sigma}{\sqrt{n_{sim}}}$$

The accuracy of the simulation is therefore inversely proportional to the square root of the number of trial n_{sim}. This means that to double the accuracy the number of trials have to be quadrupled.

5.3.1 Implementation in R

Algorithm 5 shows how a Monte Carlo simulation can be carried out to price an option. If an option, the risk free rate r, and the number of random variables len are given then by performing n_{sim} number of simulations to calculate the payoffs of the simulated paths the estimated value $\widehat{C}_n$ of the option given can be retrieved.

Algorithm 5 Monte Carlo simulation of European options

Require: an '`option`', the risk free yield r, the number of paths len, the number of simulations n_{sim}

1: **for** $i = 1 : n_{sim}$ **do**
2: generate Z_i
3: $S_i(T) = S(0)e^{(r-\frac{1}{2}\sigma^2)T+\sigma\sqrt{T}Z_i}$
4: $C_i = e^{-rT}\mathbf{max}(S(T) - X, 0)$
5: **end for**
6: $\widehat{C}_n = \frac{C_1+\ldots+C_n}{n}$

Example 5.2 shows the implementation of Algorithm 5 in R. The function takes `x`, which is an object of class '`option`', the risk free rate `r`, the length of a Wiener path `path_length`, the number of simulated paths `n_paths` and the number of simulations `n_simulations` as arguments. The function returns the given object with the simulated price included.

Example 5.2 Monte Carlo simulation routine

```
Monte_Carlo_simulation <- function(x, r, path_length, n_paths,
                                   n_simulations=50, antithetic=TRUE){
  if(!inherits(x, "option")) stop("'x' must be of class 'option'")
  price <- vector()
  dt <- maturity(x)/path_length
  mu <- underlying(x)[1]
  sigma <- underlying(x)[2]
  for( i in 1:n_simulations){
    eps <- matrix(rnorm(path_length*n_paths),nrow=path_length)

    if(antithetic)
      eps <- cbind(eps, -eps)
    path <- (mu - sigma^2/2)*dt + sigma*sqrt(dt)*eps
    payoffs <- apply(path,2,payoff,x=x,r=r)
    price[i] <- mean(payoffs)
  }
```

```
  price_path(x) <- price
  price_of(x) <- mean(price)
  x
}
```

Recalling the option from Example 5.1 we can now use the function `Monte_Carlo_simulation()` to estimate the price of the option. This is shown in Example 5.3 using a path length of 30 steps and 50 simulation runs. In each simulation run the mean of 5000 simulated payoffs is calculated. Eventually the mean of the 50 estimated option prices and is stored in the object (see also the routine from Example 5.2).

Example 5.3 Estimating the price of an European option

```
> priced <- Monte_Carlo_simulation(european, r = 0.1,
+     path_length = 30, n_paths = 5000)
> priced

A Call option with strike price 100
expiring in 30 days
Call price:  5.00535330924775
```

Table 5.1 shows estimates of the option price for three different number of simulation runs and the corresponding runtime. Furthermore, the confidence interval of the estimate is calculated. It can be seen that even with a low number of trials it takes more than 10 seconds to finish the simulation. However, the standard error of the simulation is rather small for all results. The difference of the standard error between 50 and 150 trials is negligible small.

The Convergence of the option price when increasing the number of simulations can be seen in Figure 5.3. The grey line represents the Black-Scholes price of the option.

	#Sim	MC estimate	CI−	CI+	Runtime[s]
1	10.00	5.02	4.99	5.04	10.82
2	50.00	5.01	5.00	5.03	53.83
3	150.00	5.01	5.00	5.02	161.14

Table 5.1: Monte Carlo estimates and confidence intervals

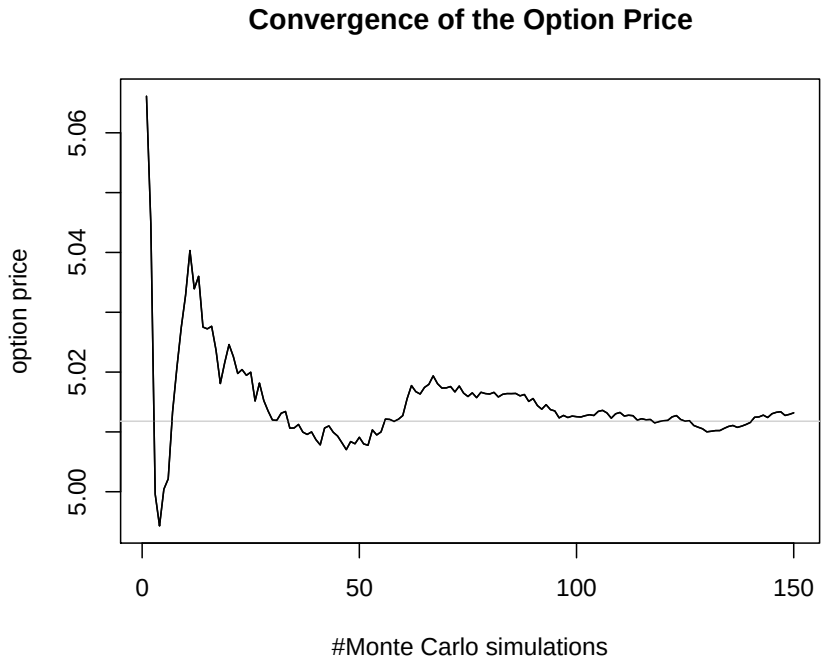

Figure 5.3: Convergence of the option price with respect to the number of Monte Carlo simulations

5.3.2 Parallel Monte Carlo Simulation

In this thesis we implemented a parallel version of Monte Carlo simulation in R using the package **Rmpi**.

Example 5.4 shows the implementation of the master process in R. This routine is responsible for process management, data distribution and collection of the simulation results. First the necessary parameters for the simulation are distributed among the slaves. Then the matrix multiplication is initialized (corresponding slave program can be seen in Example 5.5). Eventually results (all the prices calculated) are gathered and returned.

Example 5.4 MPI master routine

```
mcs.Rmpi <- function(x, r, path_length, n_paths, n_simulations=50,
                     antithetic = TRUE, n_cpu = 1,
                     spawnRslaves=FALSE) {
  if(!inherits(x,"option")) stop("'x' must be of class 'option'")
  if( n_cpu == 1 )
    return(Monte_Carlo_simulation(x, r, path_length, n_paths,
```

```
                                    n_simulations, antithetic))
  if(spawnRslaves)
    mpi.spawn.Rslaves(nslaves = n_cpu)
  ## broadcast data and functions necessary on slaves
  mpi.bcast.Robj2slave(x)
  mpi.bcast.Robj2slave(r)
  mpi.bcast.Robj2slave(path_length)
  mpi.bcast.Robj2slave(n_paths)
  mpi.bcast.Robj2slave(antithetic)
  nsim <- ceiling(n_simulations/n_cpu)
  nsim_on_last <- n_simulations - (n_cpu - 1)*nsim
  mpi.bcast.Robj2slave(nsim)
  mpi.bcast.Robj2slave(nsim_on_last)
  mpi.bcast.Robj2slave(mcs.Rmpi.slave)
  ## start partial matrix multiplication on slaves
  mpi.bcast.cmd(mcs.Rmpi.slave())
  ## gather partial results from slaves
  local_prices <- NULL
  prices <- mpi.gather.Robj(local_prices, root=0, comm=1)
  if(spawnRslaves)
    mpi.close.Rslaves()
  prices
}
```

The slave routine is basically the same like the serial version from Example 5.2 except that it returns the local estimated prices to the master slave.

Example 5.5 MPI slave routine

```
mcs.Rmpi.slave <- function(){
  require("paRc")
  commrank <- mpi.comm.rank() - 1
  Monte_Carlo_slave <- function(x, r, path_length, n_paths,
                                n_simulations=50, antithetic=TRUE){
    price <- vector()
    dt <- maturity(x)/path_length
    mu <- underlying(x)[1]
    sigma <- underlying(x)[2]
    for( i in 1:n_simulations){
      eps <- matrix(rnorm(n*length),nrow=path_length)

      if(antithetic)
        eps <- cbind(eps, -eps)
```

```
      path <- (mu - sigma^2/2)*dt + sigma*sqrt(dt)*eps
      payoffs <- apply(path,2,payoff,x=x,r=r)
      price[i] <- mean(payoffs)
    }
    price
  }
  if(commrank==(n_cpu - 1))
    local_prices <- Monte_Carlo_slave(x, r, path_length, n_paths,
                                      nsim_on_last, antithetic)
  else
    local_prices <- Monte_Carlo_slave(x, r, path_length, n_paths,
                                      nsim, antithetic)
  mpi.gather.Robj(local_prices, root=0, comm=1)
}
```

5.3.3 Notes on Parallel Pseudo Random Number Generation

Prior generating random numbers in parallel, a parallel pseudo random number generator (PPRNG) has to be initialize (e.g., SPRNG is available in R via package **rsprng**). These generators assure that independend streams of pseudo random numbers are available on the slaves (i.e., streams generated on one node are independend and uncorrelated to streams generated on all of the other nodes). This is a crucial part for the quality of Monte Carlo estimates. Furthermore, the period length of generated random number streams has to be long enough to cover the most sophisticated simulations and the generator itself has to be fast enough to be usable in high performance computing.

As already mentioned in Section 3.7 interfaces to widely used PPRNG are available in R (packages **rsprng** and **rlecuyer**).

5.3.4 Results of Parallel Monte Carlo Simulation

The results shown in this section were produced on a shared memory machine using up to 4 cores (bignodes of cluster@WU) and on a distributed memory platform using up to 10 cluster nodes. We used 50 simulation runs to estimate the option prices. The corresponding benchmark script can be found in Appendix C.

Figure 5.4 and Table 5.2 show that a good speedup can be achieved on a shared memory machine. Furthermore, in comparison to the results of the parallel matrix multiplication presented in Section 4.5 a better scalability is

	#CPU	Type	Time	Speedup
1	1	normal	53.87	1.00
2	1	MPI	53.69	1.00
3	2	MPI	27.53	1.96
4	3	MPI	19.84	2.72
5	4	MPI	15.70	3.43

Table 5.2: Time and speedup achieved on a shared memory platform using 50 simulation runs

	#CPU	Type	Time	Speedup
1	1	normal	50.45	1.00
2	1	MPI	50.30	1.00
3	2	MPI	27.28	1.85
4	3	MPI	18.62	2.71
5	4	MPI	14.87	3.39
6	5	MPI	11.56	4.36
7	6	MPI	10.49	4.81
8	7	MPI	9.46	5.33
9	8	MPI	8.38	6.02
10	9	MPI	7.27	6.94
11	10	MPI	6.19	8.16

Table 5.3: Time and speedup achieved on cluster@WU using 10 nodes and 50 simulation runs

observed for both, using the shared memory paradigm and using the distributed memory paradigm (Figure 5.5 and Table 5.3). This is because only few data is sent through the interconnection network and therefore (with Amdahl's law in mind) a greater fraction of the whole work can be executed in parallel. Moreover, when using the distributed memory paradigm only a slight overhead is observed in comparison to the shared memory paradigm because again of the fact that we have low communication costs.

To sum up, a substantial reduction of the execution time can be achieved when taking advantage of parallel computing. In fact we reduced the runtime of the simulation from over 50 seconds necessary with one processor to nearly 6 seconds using 10 processors (see Table 5.3).

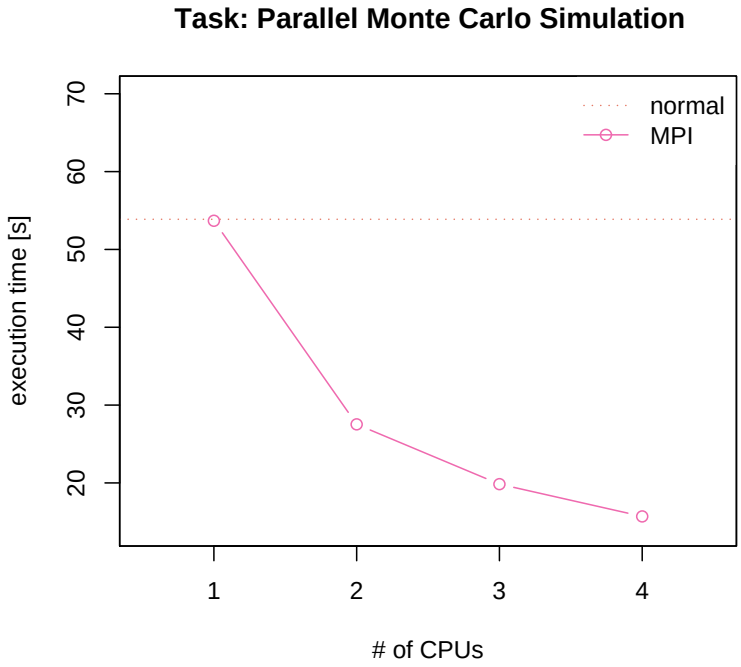

Figure 5.4: Performance of parallel Monte Carlo simulation using 50 simulation runs on a shared memory machine

5.4 Conclusion

In this chapter we presented a possible application of parallel computing in computational finance–a field with an increasing demand for high performance computing.

Parallel Monte Carlo simulation makes extensive use of parallel random number generators and therefore they have a major impact on the overall performance. Further research is promising in this area as only few alternatives exist. Furthermore, Monte Carlo simulation is not only an issue in finance but also for other problems which can only be solved numerically.

All in all in this case study we have shown that high scalability can be achieved for this type of application as work can nearly be perfectly divided among the slaves and only few data has to be sent via the network. Results showed good speedups with increasing CPU counts.

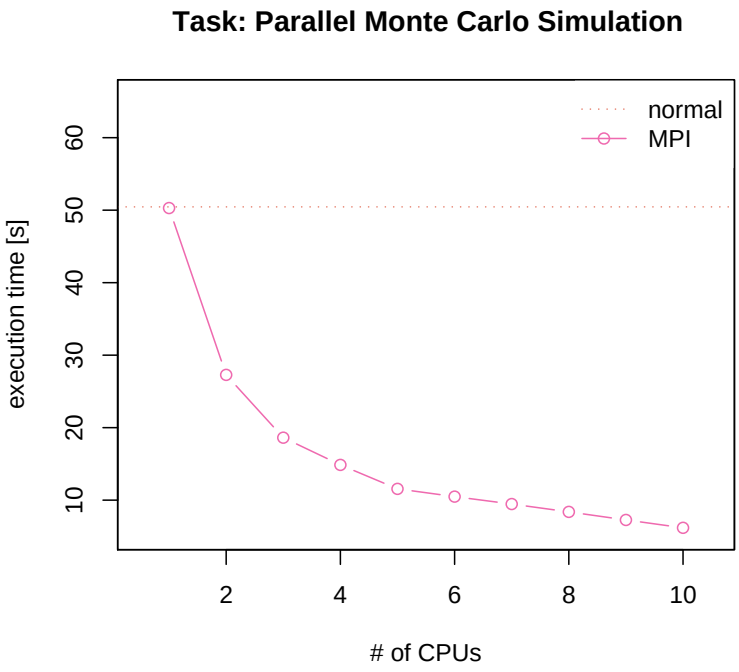

Figure 5.5: Performance of parallel Monte Carlo simulation using 50 simulation runs on cluster@WU using 10 nodes

Chapter 6

Conclusion and Future Work

6.1 Summary

The overview of the field of high performance computing presented in the first chapters is a good start to become familiar with the state of the art in high performance computing and in particular in parallel computing. Furthermore we presented in this thesis all relevant packages which extent the base R environment with parallel computing facilities. All in all they are a good start to implement parallel routines although it is hard work to get routines running with high stability. Nevertheless as algorithms become more time consuming and problems in computational statistics become more data intensive we propose to have at least a look on the possibilities which are offered from parallel programming models.

With the introduction of OpenMP parallel programming has become easier than ever. In this thesis we presented how one can only with a few statements parallelize existing serial code to achieve a significant reduction of the runtime. With R coming the restriction to low level languages like C or FORTRAN is a disadvantage. Therefore we propose implementing high level interface functions to efficient parallel OpenMP low level routines.

Using the benchmark environment provided by the package **paRc** shows that high level functions designed for general use in parallel computing (like the routines in **snow**) indeed help the user to achieve parallelism easily but with the drawback of lower performance compared to specialized functions.

The second contribution of this thesis is the parallel implementation of Monte Carlo simulation with a special focus on derivative pricing. Especially in finance, with the increasing importance of algorithmic trading, minimizing time is of major interest. The need for cutting edge hardware of investment companies and banks to minimize round tripping underlines this trend. Using

high performance computing is becoming increasingly important in this field as parallel computers become mainstream

6.2 Economical Coherences

The economic principle has ever been maximizing profits. This means either raising of revenue, increasing the return or augmenting assets. In today's ever faster society time has become an important figure. And indeed, minimizing time to complete a task can be seen as some sort of profit. In view of finance e.g., saving time for calculating a trading strategy means an advance in knowledge and therefore can raise ones (monetary) profits as investors may respond earlier to market movements or as arbitrage earnings could be generated.

In business administration optimization plays a major role. Actually in cost accounting and controlling linear problems are solved using linear programming (e.g., to determine economic order quantities). In transportation one can think of automatic shortest path selection. Selecting an optimal path is of major interest as it would maximize the deliveries possible in a certain time frame or prevent empty drives of trucks for example. The classical example for this type of problems is the "travelling salesperson problem". Nevertheless, optimization is expensive in terms of computational time and therefore it is desirable to have efficient (parallel) algorithms available. Parallel computing would be a possibility to achieve higher performance in optimization.

Concluding we would like to say that we live in an economy which depends more and more on information technology. In the near future many IT companies will have to take the step from the serial to the parallel world otherwise they will not benefit from the recent advances in this area (and the research done in this area is even expected to increase). This also means that a major rethinking will have to take place in software development.

Nevertheless taking advantage of parallelism through dividing a problem in smaller (specialized) tasks is not a new idea in economics. Already a long time ago one of the most important economists Smith (1776) proposed the division of labor to increase welfare. Now, it is time to divide tasks such that computational time is minimized or in other words profit is increased.

6.3 Outlook

High performance computing is applied to these tasks which are complex to solve and therefore need huge amount of computational time or for time critical tasks where it is important to save every millisecond.

Today high performance computing is used in many scientific areas where sophisticated computations are carried out. An important field which can be applied in many fields in economics is optimization. $\mathcal{NP}$-hard problems like decision problems, search problems or as mentioned optimization problems are solved in polynomial time. With parallel computing time could be reduced to solve these problems provided that the problems can be subdivided accordingly.

In this thesis classical examples like the matrix multiplication in parallel computing were shown. A lot of research has been done in this area to provide even faster algorithms. A next step would be to implement the most promising variants (like the PUMMA) of them to become familiar with sophisticated parallel programming algorithms.

Many improvements are possible for package **paRc**. The benchmark environment could be enhanced to provide greater flexibility to the user so that new benchmark alternatives can easily be integrated. Furthermore, new parallel routines could be developed using the OpenMP programming model as an interface to the OpenMP library already exists.

Further research could clearly be done in the computational finance segment. The recent advances in algorithmic trading and the increasing need for computational power clearly indicate a trend towards high performance computing. The option pricing environment in package **paRc** supplies a framework for pricing derivatives using parallel Monte Carlo simulation which makes heavily use of parallel random number generators. A next step would be to improve parallel pseudo random number generation. Only few research has been done in this area in the last years.

Besides the topics mentioned above, many more could be thought of and implemented, thus there are more than enough fields for further research.

6.4 Acknowledgments

I was extremely lucky to carry out my diploma thesis at the Department of Statistics and Mathematics. My position as a studies assistant gave me the opportunity to discuss many topics of this thesis with the staff members—my colleagues. Especially I would like to thank my supervisor Kurt Hornik for his guidance and mentorship and of course for the many reading.

I would like to thank David Meyer as well as Achim Zeileis who helped me with statistical, graphical and information technology issues.

Furthermore I would like to thank Josef Leydold for his hints regarding random number generation.

Last but not least, I would like to express my thanks to my parents for their ongoing encouragement. In addition I would like to thank my girlfriend Selma for proofreading and of course for her patience and her support.

Appendix A

Installation of Message Passing Environments

This appendix shows how one can set up an appropriate message passing environment and how this environment can be handled.

A.1 LAM/MPI

For installing LAM/MPI on a workstation or a cluster either the source code or pre-compiled binaries can be downloaded from the project website (`http://www.lam-mpi.org/`). The source code can be compiled as follows (job script for SGE):

```
#$ -N compile-lam-gcc

## change to the source directory of downloaded LAM/MPI
cd /path/to/lamsource/
echo "#### clean ####"
make clean
echo "#### configure ####"
## configure - enable shared library support
./configure CC=gcc CXX=g++  FC=gfortran \
  --prefix=/path/to/lamhome --enable-shared
echo "#### make ####"
make all
echo "#### install ####"
make install
echo "#### finished ####"
```

If Debian Etch is used LAM/MPI can be installed using `apt-get install` followed by the packages

lam-runtime LAM runtime environment for executing parallel programs,

lam4-dev Development of parallel programs using LAM,

lam4c2 Shared libraries used by LAM parallel programs.

To boot the LAM/MPI environment one can simply call `lamboot` from the command line. This sets up a 1 node "parallel" environment. To have more nodes connected a configuration file has to be specified.

```
#
# example LAM host configuration file
# bhost.conf
#
#server.cluster.example.com schedule=no
localhost cpu=4
```

With this file given executing `lamboot bhost.conf` sets up a parallel environment with 4 nodes on localhost.

A.2 PVM

Building and installing the PVM daemon from source is explained extensively in the book of Geist et al. (1994).

If Debian Etch is used PVM can be installed using `apt-get install` followed by the packages

pvm Parallel Virtual Machine - binaries

pvm-dev Parallel Virtual Machine - development files

libpvm3 Parallel Virtual Machine - shared libraries

Appendix B

Sun Grid Engine

A detailed documentation is available from the Sun Microsystems website (Sun Microsystems, Inc. (2007)). In this appendix the fundamental steps for running parallel routines with R are explained.

Important for running applications with the grid engine are job scripts. They work in the same way like shell scripts but additionally contain control sequences for the grid engine. These control sequences always start with `#$` followed by a command flag and an argument.

Often used flags are:

`-N` defines that the following argument is the name of the job.

`-q` defines the queue to use.

`-pe` defines which parallel environment is to be started and how many nodes have to be reserved.

B.1 LAM/MPI Parallel Environment

To use the LAM/MPI parallel environment (pe) with the grid engine the following has to be inserted into the user's .bashrc:

```
export LD_LIBRARY_PATH=/path/to/lamhome/lib:$LD_LIBRARY_PATH
export LAMHOME=/path/to/lamhome
```

where the paths point to the corresponding LAM/MPI installation.

Cluster Job Scripts

A job script for starting a LAM/MPI application looks like this:

```
#$ -N Rmpi-example
# use parallel environment lam with 12 nodes
#$ -pe lam 12
# run job on queue node.q to have a homogeneous processing
# environment
#$ -q node.q

R --vanilla < /path/to/my/Rmpi-example.R
```

This job of the name "Rmpi-example" starts a parallel LAM/MPI environment with 12 nodes on `node.q` of cluster@WU. Eventually R is called sourcing a specific file containing parallel R code.

B.2 PVM Parallel Environment

To use the PVM pe with the grid engine the following has to be inserted into the user's .bashrc:

```
# PVM>
# you may wish to use this for your own programs (edit the last
# part to point to a different directory f.e. ~/bin/_\$PVM_ARCH
#
if [ -z $PVM_ROOT ]; then
    if [ -d /home/stheussl/lib/pvm3 ]; then
        export PVM_ROOT=/home/stheussl/lib/pvm3
    else
        echo "Warning - PVM_ROOT not defined"
        echo "To use PVM, define PVM_ROOT and
              rerun your .bashrc"
    fi
fi

if [ -n $PVM_ROOT ]; then
    export PVM_ARCH=‘$PVM_ROOT/lib/pvmgetarch‘
```

```
# uncomment one of the following lines if you want the PVM
# commands directory to be added to your shell path.
#
       export PATH=$PATH:$PVM_ROOT/lib           # generic
       export PATH=$PATH:$PVM_ROOT/lib/$PVM_ARCH # arch-specific
#
# uncomment the following line if you want the PVM
# executable directory to be added to your shell path.
#
       export PATH=$PATH:$PVM_ROOT/bin/$PVM_ARCH
fi
```

Cluster Job Scripts

A job script for starting a PVM application looks like this:

```
#$ -N rpvm-test
# use parallel environment pvm with 8 nodes
#$ -pe pvm 8
# run job on queue node.q to have a homogeneous processing
# environment
#$ -q node.q

R --vanilla < /path/to/my/rpvm-example.R
```

This job of the name "rpvm-test" starts a parallel PVM environment with 12 nodes on `node.q` of cluster@WU. Eventually R is called sourcing a specific file containing parallel R code.

Appendix C

Benchmark Definitions

In this appendix the benchmark source files and job scripts are shown.

C.1 Matrix Multiplication

Uniformly Distributed Matrices

In general we used the uniform distribution ($[-5, 5]$) to run the benchmark in this thesis. The code for running a benchmark on a distributed memory machine and a detailed description can be found in Section 4.4.1. The following code describes a benchmark for a shared memory machine.

```
## load required libraries
library("Rmpi")
library("paRc")
library("snow")

## definition of benchmark
max_cpu <- 4
task <- "matrix multiplication"
taskID <- "mm"
paradigm <- "shared"
types <- c("OpenMP", "MPI","snow-MPI", "MPI-wB")
complexity <- c(1000, 2500)
runs <- 250
## data description
bmdata <- list()
bmdata[[1]] <- bmdata[[2]] <- 1000
bmdata[[3]] <- function(x){
  runif(x,-5,5)
}
```

```
## create benchmark object
bm <- create_benchmark(task=task, data=bmdata,
                       type=types[1], parallel=TRUE,
                       cpu_range=1:max_cpu, runs=runs)
set.seed(1782)
## for each complexity and type run a number of benchmarks
for(n in complexity){
  bmdata[[1]] <- bmdata[[2]] <- n
  benchmark_data(bm) <- bmdata
  for(type in types){
    benchmark_type(bm) <- type
    writeLines(paste("Starting",type,"benchmark with complexity",n,
                     "..."))
    results <- run_benchmark(bm)
    save(results,file=sprintf("%s-%s-%s-%d.Rda", paradigm, taskID,
                              type, n))
  }
}
```

and the corresponding cluster job script looks as follows:

```
#$ -N MPI-distr-MM
#$ -pe lam 20
#$ -q node.q

/path/to/R-binary --vanilla < \
  /path/to/MPI-benchmark-examples/distributed-mm.R
```

Normally Distributed Matrices

When using normally distributed entries in the matrices we used the following benchmark definition to run on a distributed memory machine:

```
## load required libraries
library("rpvm")
library("paRc")
library("snow")

## definition of benchmark run
max_cpu <- 20
task <- "matrix multiplication"
taskID <- "mm-norm"
paradigm <- "distributed"
```

```
types <- c("PVM","snow-PVM")
complexity <- c(1000, 2500, 5000)
runs <- 250
bmdata <- list()
bmdata[[1]] <- bmdata[[2]] <- 1000
bmdata[[3]] <- function(x){
  rnorm(x)
}

bm <- create_benchmark(task=task, data=bmdata,
                       type=types[1], parallel=TRUE,
                       cpu_range=1:max_cpu, runs=runs)
set.seed(1782)
for(n in complexity){
  bmdata[[1]] <- bmdata[[2]] <- n
  benchmark_data(bm) <- bmdata
  for(type in types){
    benchmark_type(bm) <- type
    writeLines(paste("Starting",type,"benchmark with complexity",n,
                     "..."))
    results <- run_benchmark(bm)
    save(results,file=sprintf("%s-%s-%s-%d.Rda", paradigm, taskID,
                              type, n))
  }
}
```

and the corresponding job script for the grid engine looks as follows:

```
#$ -N PVM-norm-distr-MM
#$ -pe pvm 20
#$ -q node.q

/path/to/R-binary --vanilla < \
  /path/to/PVM-benchmark-examples/distributed-mm.R
```

C.2 Option Pricing

To benchmark parallel Monte Carlo simulations the following code was used:

```
## load required libraries
library("Rmpi")
library("paRc")
```

```
## definition of benchmark run
max_cpu <- mpi.universe.size()
task <- "Monte Carlo simulation"
taskID <- "mcs"
paradigm <- "distributed"
types <- c("normal", "MPI")
runs <- 10

bm <- create_benchmark(task=task, data=list(),
                       type=types[1], paralle=TRUE,
                       cpu_range=1:max_cpu, runs=runs)
## define option
opt <- define_option(c(0.1,0.4,100),100,1/12)

bmdata <- list()
bmdata[[1]] <- opt
bmdata[[2]] <- 0.1 ## yield
bmdata[[3]] <- 30  ## path length
bmdata[[4]] <- 5000 ## number of paths
bmdata[[5]] <- 50  ## number of simulations
bmdata[[6]] <- TRUE ## use antithetic
benchmark_data(bm) <- bmdata

for(type in types){
  benchmark_type(bm) <- type
  writeLines(paste("Starting",type,"benchmark..."))
  results <- run_benchmark(bm)
  save(results,file=sprintf("%s-%s-%s-%d.Rda", paradigm, taskID,
                            type, n))
}
```

the corresponding job script looks as follows:

```
#$ -N MPI-distr-MCS
#$ -pe lam 10
#$ -q node.q

/path/to/R-binary --vanilla < \
  /path/to/MPI-benchmark-examples/distributed-mcs.R
```

List of Figures

2.1 Amdahl's Law for parallel computing 28

3.1 Payoff of a European call option with price 5.93 58

4.1 Comparison of parallel programming models on a shared memory machine and complexity grade 1000 77
4.2 Comparison of parallel programming models on a shared memory machine using BLAS and complexity grade 1000 79
4.3 Comparison of parallel programming models on a shared memory machine and complexity grade 2500 80
4.4 Comparison of parallel programming models on a shared memory machine with BLAS and complexity grade 2500 82
4.5 Comparison of parallel programming models on a cluster using 20 nodes and complexity grade 2500 83
4.6 Comparison of parallel programming models on a cluster using 20 nodes using BLAS and complexity grade 2500 84
4.7 Comparison of parallel programming models on a cluster using 20 nodes and complexity grade 5000 85
4.8 Comparison of parallel programming paradigms on a cluster using 20 nodes with BLAS and complexity grade 5000 86

5.1 Payoffs of a European put and call option depending on the position of the investor 90
5.2 Path of a Wiener process 91
5.3 Convergence of the option price with respect to the number of Monte Carlo simulations 99
5.4 Performance of parallel Monte Carlo simulation using 50 simulation runs on a shared memory machine 103
5.5 Performance of parallel Monte Carlo simulation using 50 simulation runs on cluster@WU using 10 nodes 104

List of Tables

2.1 Interleaved concurrency 6
2.2 Parallelism 6
2.3 Pipelining 10
2.4 cluster@WU specification 29
2.5 Opteron server specification 29

4.1 Comparison of BLAS routines on a bignode with complexity grade 1000 65
4.2 Comparison of BLAS routines on a bignode with complexity grade 2500 65
4.3 Comparison of parallel programming models on a shared memory machine and complexity grade 1000 76
4.4 Comparison of parallel programming models on a shared memory machine using BLAS and complexity grade 1000 78
4.5 Comparison of parallel programming models on a shared memory machine and complexity grade 2500 78
4.6 Comparison of parallel programming models on a shared memory machine with BLAS and complexity grade 2500 81
4.7 Comparison of parallel programming models on a cluster using 20 nodes and complexity grade 2500 81
4.8 Comparison of parallel programming models on a cluster using 20 nodes and complexity grade 5000 82

5.1 Monte Carlo estimates and confidence intervals 98
5.2 Time and speedup achieved on a shared memory platform using 50 simulation runs 102
5.3 Time and speedup achieved on cluster@WU using 10 nodes and 50 simulation runs 102

Bibliography

Gene Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS Conference Proceedings*, 30(8):483–485, 1967.

D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

Louis Bachelier. *Théorie de la Spéculation*. PhD thesis, Gauthier-Villars, 1900.

D.J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawak, and C.V. Packer. Beowulf: A Parallel Workstation for Scientific Computation. *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, pages 11–14, 1995.

Fischer Black and Myron Scholes. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. *Proceedings of Supercomputer Symposium '94*, 94:379–386, 1994.

J.M. Chambers and T.J. Hastie. *Statistical Models in S*. CRC Press, Inc. Boca Raton, FL, USA, 1991.

Jaeyong Choi, Jack Dongarra, and David Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on Distributed Memory Concurrent Computers. Technical report, ORNL/TM–12252, Oak Ridge National Lab., TN (United States), 1993.

Duane Currie. *papply: Parallel apply Function Using MPI*, 2005. URL `http://ace.acadiau.ca/math/ACMMaC/software/papply/`. R package version 0.1.

L. Dagum. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. *OpenMP. org*, 1997.

J.J. Dongarra, LS Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, et al. ScaLAPACK User's Guide. *Society for Industrial and Applied Mathematics (SIAM)*, 1997.

R. Duncan. A Survey of Parallel Computer Architectures. *Computer*, 23(2): 5–16, 1990.

Michael Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

Ian Foster and Carl Kesselman. Globus: a Metacomputing Infrastructure Toolkit. *International Journal of High Performance Computing Applications*, 11(2):115, 1997.

Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

G.C. Fox, S.W. Otto, and A.J.G. Hey. Matrix Algorithms on a Hypercube I: Matrix Multiplication. *Parallel Computing*, 4:17–31, 1987.

E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, 2004.

Narain Gehani and Andrew McGettrick, editors. *Concurrent Programming*. Addison-Wesley Publishing Company, 1988.

Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

GA Geist, JA Kohl, and PM Papadopoulos. PVM and MPI: a Comparison of Features. *Calculateurs Paralleles*, 8(2):137–150, 1996.

W. Gentzsch et al. Sun Grid Engine: Towards Creating a Compute Power Grid. *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

Paul Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins University Press Baltimore, MD, USA, 1996.

Kazushige Goto. *GotoBLAS FAQ*, 2007. URL `http://www.tacc.utexas.edu/resources/software/gotoblasfaq.php`.

W. Gropp and E. Lusk. Goals Guiding Design: PVM and MPI. *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 257–265, 2002.

W. Gropp, E.L. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

Gruppe Deutsche Börse. Deutsche Börse vervierfacht Netzwerkbandbreite für Xetra, 2007a. URL `http://www.deutsche-boerse.com`. Press release.

Gruppe Deutsche Börse. Deutsche Börse bietet erweiterte Xetra-Handelsdaten mit geringerer Latenz, 2007b. URL `http://www.deutsche-boerse.com`. Press release.

John Hennessy and David Patterson, editors. *Computer Architecture—A Quantitative Approach*. Morgan Kaufmann, 2007.

Jay Hoeflinger. *Extending OpenMP* to Clusters*. Intel Corporation, 2006. URL `http://www.intel.com`. White Paper.

Kurt Hornik. The R FAQ, 2007. URL `http://CRAN.R-project.org/doc/FAQ/R-FAQ.html`. ISBN 3-900051-08-9.

John C. Hull. *Options, Futures and Other Derivatives*. Prentice Hall, 5 edition, 2003.

R. Ihaka and R. Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

Intel Corporation. *Intel C++ Compiler Documentation*. Intel Corporation, 2007a. URL `www.intel.com`.

Intel Corporation. *Intel Fortran Compiler Documentation*. Intel Corporation, 2007b. URL `www.intel.com`.

Intel Corporation. *Intel Math Kernel Library for Linux User's Guide*, 2007c. URL `http://developer.intel.com`.

Kiyoshi Itô. *On Stochastic Differential Equations*. American Mathematical Society, 1951.

J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy. *Introduction to the Cell Multiprocessor*, 2005. URL `http://www.ibm.com`.

Jason Kelly. Hal 9000-style machines, Kubrick's fantasy, outwit traders, 2007. URL `http://www.bloomberg.com`.

Andreas Kleen. *A NUMA API for Linux*. Novell, Inc., 2005. URL `http://www.novell.com`. White Paper.

Erricos Kontoghiorghes, editor. *Handbook of Parallel Computing and Statistics*. Chapman & Hall, 2006.

E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky. SETI@Home—Massively Distributed Computing for SETI. *Computing in Science & Engineering [see also IEEE Computational Science and Engineering]*, 3(1):78–83, 2001.

CL Lawson, RJ Hanson, DR Kincaid, and FT Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

Peter Lazar and David Schoenfeld. *biopara: Self-contained Parallel System for R*, 2006. R package version 1.4.

P. L'Ecuyer, R. Simard, E.J. Chen, and W.D. Kelton. An Object-Oriented Random-Number Package With Many Long Streams and Substreams. *Operations Research*, 50(6), 2002.

Na (Michael) Li. *rsprng: R Interface to SPRNG (Scalable Parallel Random Number Generators)*, 2007. URL `http://www.r-project.org/,http://www.biostat.umn.edu/~nali/SoftwareListing.html`. R package version 0.3-4.

Na (Michael) Li and Anthony Rossini. *rpvm: R Interface to PVM (Parallel Virtual Machine)*, 2007. URL `http://www.r-project.org/`. R package version 1.0.2.

Greg MacSweeney. Pleasures and Pains of Cutting-Edge Technology, 2007. URL `http://www.wstonline.com`.

M. Mascagni and A. Srinivasan. SPRNG: A Scalable Library for Pseudo Random Number Generation. *ACM Transactions on Mathematical Software*, 26(3):436–461, 2000.

Robert Merton. Theory of Rational Option Pricing. *Bell Journal of Economics and Management Science*, 4:141–183, 1973.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1994.

Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, 2003.

Gordon Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), 1965.

J. Nieplocha, R.J. Harrison, and R.J. Littlefield. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 2.5*, 2005.

Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

R Development Core Team. *Writing R Extensions*. R Foundation for Statistical Computing, Vienna, Austria, 2007a. URL `http://www.R-project.org`.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2007b. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

Anthony Rossini, Luke Tierney, and Na Li. Simple Parallel Statistical Computing in R. *UW Biostatistics Working Paper Series*, (Working Paper 193), 2003. URL `http://www.bepress.com/uwbiostat/paper193`.

Nagiza Samatova, Srikanth Yoginath, David Baues, and Guruprasad Kora. *A Seamless Interface to Perform Parallel Computation on Linear Algebra Problems Using the ScaLAPACK Library*, 2005. URL `http://www.aspect-sdm.org/Parallel-R`. R package version 0.5.1.

Nagiza F. Samatova, David Bauer, and Srikanth Yoginath. *taskPR: Task-Parallel R Package*, 2004. URL `http://www.aspect-sdm.org/Parallel-R/`. R package version 0.2.7.

N.F. Samatova, M. Branstetter, A.R. Ganguly, R. Hettich, S. Khan, G. Kora, J. Li, X. Ma, C. Pan, A. Shoshani, et al. High Performance Statistical Computing with Parallel R: Applications to Biology and Climate Modelling. *Journal of Physics*, pages 505–509, 2006.

Hana Sevcikova. Statistical Simulations on Parallel Computers. *Journal of Computational and Graphical Statistics*, (13):886–906, 2004.

Hana Sevcikova and A. J. Rossini. *snowFT: Fault Tolerant Simple Network of Workstations*, 2004a. R package version 0.0-2.

Hana Sevcikova and A.J. Rossini. Pragmatic Parallel Computing. *Journal of Statistical Software*, 2004b.

Hana Sevcikova and Tony Rossini. *R Interface to RNG with Multiple Streams*, 2004c. R package version 0.1.

Adam Smith. *An Inquiry Into the Nature and Causes of the Wealth of Nations*. 1776.

Richard Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*. The Free Software Foundation, 2007. URL `http://gcc.gnu.org`.

Sun Microsystems, Inc. *Sun N1 Grid Engine 6.1 User's Guide*. Sun Microsystems, Inc., Santa Clara, CA U.S.A., 2007. URL `http://docs.sun.com/app/docs/doc/820-0699?a=load`.

Stefan Theussl. *R-Forge User's Manual*, 2007a. URL `http://download.r-forge.r-project.org/manuals/R-Forge_Manual.pdf`.

Stefan Theussl. *paRc: paRallel Computations in R*, 2007b. URL `http://parc.r-forge.r-project.org`. R package version 0.0-9.

Luke Tierney, Anthony Rossini, Na Li, and H. Sevcikova. *snow: Simple Network of Workstations*, 2007. R package version 0.2-3.

Wikipedia. High performance computing — wikipedia, the free encyclopedia, 2007a. URL `http://en.wikipedia.org/w/index.php?title=High-performance_computing&oldid=151017056`. [Online; accessed 14-August-2007].

Wikipedia. Computational overhead — wikipedia, the free encyclopedia, 2007b. URL `http://en.wikipedia.org/w/index.php?title=Computational_overhead&oldid=130287492`. [Online; accessed 28-August-2007].

Gregory Wilson and Paul Lu, editors. *Parallel Programming Using C++*. The MIT Press, 1996.

S. Yoginath, NF Samatova, D. Bauer, G. Kora, G. Fann, and A. Geist. RScaLAPACK: High-Performance Parallel Statistical Computing with R and ScaLAPACK. *Proceedings of the 18th International Conference on Parallel and Distributed Computing Systems*, pages 12–14, 2005.

Hao Yu. Rmpi: Parallel Statistical Computing in R. *R News*, 2(2):10–14, 2002.

Hao Yu. *Rmpi: Interface (Wrapper) to MPI (Message-Passing Interface)*, 2006. URL `http://www.stats.uwo.ca/faculty/yu/Rmpi`. R package version 0.5-3.