CLUE — Cluster Evaluation

Helmut Hlavacs[1]
Dieter F. Kvasnicka[2]
Christoph W. Ueberhuber[3]

[1]Institute for Computer Science and Business Informatics,
University of Vienna
hlavacs@aurora.tuwien.ac.at

[2]Institute for Physical and Theoretical Chemistry,
Technical University of Vienna
kvasnicka@tuwien.ac.at

[3]Institute for Applied and Numerical Mathematics,
Technical University of Vienna
christof@uranus.tuwien.ac.at

March 2000

AURORA TR2000-05

The work described in this report was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund.

Abstract

This report describes the simulation tool CLUE which enables the highly accurate performance assessment and prediction of clusters of symmetric multiprocessors (SMPs). Using CLUE, reliable information can be obtained to reach the optimum decision on hardware configurations (processing elements and communication networks) before actually purchasing this hardware. Thus, hardware can be adapted to individual software features, reversing the currently applied adaptation of high-performance software to hardware features (as used, for instance, in FFTW [8, 9], PHIPAC [2], or the ATLAS tool [18]).

Contents

Chapter 1

Introduction

In the last few years high-performance hardware has changed dramatically. Until quite recently, shared memory parallel vector processor (PVP) computers and distributed massively parallel processing (MPP) machines dominated the field of supercomputing. Nowadays high-performance computing has turned towards clusters of shared memory symmetric multiprocessors (SMPs). Most of these clusters are built using custom processors, as well as custom interconnection devices and memory. Accordingly, clusters of SMPs appear in a multitude of different configurations, ranging from teraflops machines to small clusters, comprising several PCs or workstations. The large number of possible configurations makes it difficult to decide which particular structure is suitable to solve a particular class of problems under certain performance requirements, and of course under given financial constraints. To aid this process of evaluating computer clusters the simulation and assessment tool CLUE (*clu*ster *e*valuator) has been developed. In this report the basic principles of CLUE and some results gathered in numerical experiments will be presented.

Synopsis

Section 1.1 describes current PC clusters, used in this report as examples of SMPs, including hardware, operating system, and software. Sections 1.2 and 1.3 give a description of the specific clusters and algorithms, used in the experimental sections of this report. Chapters 2 and 3 describe CLUE, the machine independent simulation system for clusters of SMPs. Chapter 4 shows the experimental scenarios and their results. Finally, Chapter 5 gives the conclusions of this work, and Chapter 6 provides an outlook on future work.

1.1 PC Clusters

As the performance of commodity computer and network hardware increases, and prices decrease, it becomes more and more attractive to build parallel computer systems from off-the-shelf components. The current price / performance ratio of a PC cluster is often ten times better than that of traditional supercomputers. PC clusters scale reasonably well, are easy to construct and only the hardware has to be paid for as most of the software is free.

Reasons for the success of PC clusters are the ever increasing efficiency of hardware components and at the same time the dramatic decline of prices. Meanwhile several entries of the top 500 supercomputer list[1] are clusters of PCs[2].

A PC cluster is a multi-computer architecture which can be used for parallel computations. It is a system which usually consists of one server node (the front-end PC), and a more or less arbitrary number of client nodes connected via Ethernet or some other network. PC clusters are usually built using commodity hardware components, like standard PCs capable of running Linux, standard Ethernet adapters, and switches. Usually PC clusters do not contain any proprietary hardware components and thus are trivially reproducible. Commodity software like the Linux operating system, Parallel Virtual Machine (PVM, see Geist et al. [10]) and Message Passing Interface (MPI, see Gropp et al. [11]) are used for developing parallel programs.

The server node controls the whole cluster and serves files to the client nodes. It is also the cluster's console and gateway to the outside world. Large clusters may have more than one server node, and possibly other nodes dedicated to particular tasks, for example consoles or monitoring stations.

In most cases client nodes are dumb, the dumber the better. Client nodes are configured and controlled by the server node, and do only what they are told to do. In a disk-less client configuration, client nodes don't even know their IP address or name until the server tells them what it is. In most cases client nodes do not have keyboards or monitors, or access to the rest of the world, and are accessed only via remote login or possibly a serial terminal. The nodes can be thought of as a (multi) CPU plus memory package which can be plugged into the cluster, just like a CPU or memory module can be plugged into a motherboard. PC clusters used for high-performance computing are often referred to as *Beowulf clusters*[3].

Typical configurations are, for example, a compound of several computers connected via Fast Ethernet or faster networks like Gigabit Ethernet, Myrinet or SCI (Scalable Coherent Interface).

As a form of parallel computers PC cluster fall somewhere in between MPPs (Massively Parallel Processors) and NOWs (Networks of Workstations). They profit of developments in both types of architectures. MPPs are usually larger and use faster networks. Typical problems when programming MPPs are load and data distribution, granularity and minimizing the communication overhead. Programs, which are not too fine grained, can be ported from MPPs to PC clusters and usually run there quite efficiently. When programming NOWs, algorithms are most often designed to utilize unused cycles of powerful workstations. These algorithms have to be extremely tolerant of load variations and have to use dynamic load balancing. All programs running on NOWs run at least as good on dedicated PC clusters.

[1] `http://www.netlib.org/benchmark/top500.html`

[2] `http://www.cs.sandia.gov/cplant/` `http://cnls.lanl.gov/avalon/` `http://wwwwissrech.iam.uni-bonn.de/research/projects/parnass2/`

[3] `http://www.beowulf.org`

1.2 Hardware

In this report two specific PC clusters are investigated.

1.2.1 The Vienna Cluster

The first PC cluster of the research project AURORA[4] was built as a cooperation between the Institutes for Applied and Numerical Mathematics and Physical and Theoretical Chemistry of the Vienna University of Technology. It consists of one master and five dual Pentium II slaves. The master is used as file and net server and does all the compilation work. In principle any user may access the nodes directly, but access may be restricted to certain users and/or certain nodes.

The compute slaves are dual 350 MHz Pentium II systems with 256 MB main memory, 512 KB Level 2 cache and local 4.5 GB hard discs. The server node is based on a single 350 MHz Pentium II, several hard discs (separated discs for system, users and backup) and 256 MB main memory. The internal network consists of 100 Mbit/s Ethernet cards (3COM), connected via a switch. Only the master is connected to the outside world by a second 10/100 Mbit/s Ethernet connection. Additional peripheral devices are a CD-ROM drive on the master and floppy disc drives on all six PCs.

The operating system is Linux, in the Suse 6.0 distribution. To get improved support for dual processor nodes, it was upgraded to kernel version 2.2 (currently 2.2.9).

Tools for parallelization are MPI (in the implementation of MPICH), PVM (version 3.4) and an HPF (High Performance Fortran) compiler of PGI (Portland Group Inc.). Available numerical libraries are the BLAS [4], LAPACK [1], and SCALAPACK [3], some of them in several versions to ensure compatibility with all compilers installed.

1.2.2 The Aachen Cluster

The PC cluster Siemens hpcLine consists of 16 dual processor boards using 400 MHz Pentium II, 512 KB Level 2 cache, 512 MB main memory and local 4 GB hard discs. The nodes communicate either via switched Fast Ethernet or SCI (Scalable Coherent Interface). The SCI network is configured as a two-dimensional torus.

All nodes run Red Hat Linux, kernel version 2.0.36 in SMP mode. Fast Ethernet is used for TCP/IP, and the SCI network is used by the MPI version of Scali called ScaMPI. Fortran 77, Fortran 90, HPF, C, and C++ compilers of PGI are available, as well as the GNU compilers.

[4] `http://www.vcpc.univie.ac.at/aurora/`

1.3 Algorithms

Algorithms for distributed memory parallel computers see three scenarios:

Computation bound algorithms achieve near-linear speedup (provided the work load is distributed evenly).

Communication bound algorithms do not speed up at all, since additional processors usually also increase the amount of communication to be performed.

Transition algorithms which are computation bound when running on few processors and communication bound when running on many processors.

When simulating these algorithms, different requirements exist. Simulation of computation bound algorithms does not depend significantly on the accuracy of the network simulation, since communication time is negligible. However, the accurate simulation of computing time is important for reliable simulation results.

Simulation of communication bound algorithms does not require an accurate processor simulation, since computation time is negligible. Accurate network simulation is crucial in this case, including (in many cases) contention analysis and simulation.

To find out the transition point (with increasing number of processors) between the computation bound mode and the communication bound mode reliably, both processors and network have to be modeled accurately.

Many algorithms which do not require fast communication scale very well on PC clusters. Linear algebra algorithms, however, usually require more tightly coupled hardware, due to finer grained communication patterns.

Blocking is applied in many linear algebra algorithms to improve locality of memory access on single processors, resulting in cache re-usage and therefore high performance. In parallel linear algebra, blocking is also used to get coarser grained communication patterns: Instead of sending m columns or rows of an $n \times n$ matrix separately, a blocked algorithm combines k such n-vectors to a single message. The overall amount of matrix elements which have to be transferred is still n^2, however the number of packets is reduced from n to $\lceil n/k \rceil$, which drastically reduces communication overhead, if latency of the network is a critical factor (which is often the case).

Blocking was applied, for example, in the design of the sequential linear algebra package LAPACK (see Anderson et al. [1]), which therefore dramatically outperforms its predecessors EISPACK and LINPACK. Also in its parallel version, SCALAPACK [3], blocking was applied. Both LAPACK and SCALAPACK are based on calls to the BLAS (basic linear algebra subprograms, see Lawson et al. [15] and Dongarra et al. [4, 5]), which are doing most of the computational work. The BLAS exist in highly optimized form for all major computing platforms. When there is no vendor provided BLAS library it is possible to build an individually optimized BLAS library using the ATLAS system [18] or PHIPAC [2]. SCALAPACK is also based on the BLACS (basic linear algebra communication subroutines, see Dongarra et al. [6, 7]), which are doing all the communication. The BLACS can be optimized according to hardware characteristics. However, this is not as crucial as performance optimization of the BLAS library. Portable implementations of the BLACS based on MPI (see Gropp et al. [11]) and PVM (see Geist et al. [10]) exist and are frequently used.

Various SCALAPACK routines were used to demonstrate the usefulness and the reliability of CLUE in numerical experiments:

Matrix Multiplication. The routine `PBLAS/pdgemm` is used to multiply two matrices: $C = AB$. All three matrices are $n \times n$, distributed two-dimensional in a block-cyclic way using the same block size for both dimensions and all matrices.

Cholesky Factorization. The routine `SCALAPACK/pdpotrf` is used to compute the Cholesky factorization $U^T U$ of a symmetric, positive definite matrix A. U is an upper triangular matrix. Only the elements above the matrix diagonal are accessed. A (and U) is stored two-dimensional in a block-cyclic way using the same block size for both matrix dimensions.

LU Factorization. The routine `SCALAPACK/pdgetrf` is used to compute the LU factorization PLU of a general matrix A. L is a lower triangular matrix with unit diagonal elements, U is an upper triangular matrix, and P is a permutation matrix, resulting from column pivoting. The output matrices U and L are stored in place of the input matrix A which is distributed in a two-dimensional block-cyclic manner using the same block size for both matrix dimensions.

Chapter 2

Features of CLUE

CLUE is based on MISS-PVM [14], the *Machine Independent Simulation System for PVM 3*. MISS-PVM enables the development of software for parallel computers which are not yet available in reality. MISS-PVM can also be used as a tool to carry out performance assessments using existing programs which are independent of actual load characteristics. MISS-PVM also makes the debugging of parallel programs easier. To exploit these features, it is not necessary to rewrite existing code or create additional one (neither in C nor in Fortran). PVM based code can be used without modification.

2.1 Overview

The features mentioned above are obtained by establishing a virtual layer which does not have to be tampered with when developing software. The *Virtual Layer for PVM 3* is situated between the user program and PVM 3 (see Figure 2.2). All calls to PVM 3 are redirected to MISS-PVM subroutines, which perform certain virtual timings and virtual machine adaptations and eventually pass on the calls (in a modified form) to PVM 3.

In order to use the virtual layer it is not necessary to modify user programs. It is only necessary to use different include files and to link the user programs to additional libraries. The virtual layer generates output files which trace calls to communication subroutines. These trace files are the inputs of post-mortem visualization.

This method has two major advantages over normal trace file writing: the *Virtual Layer for PVM 3* (i) uses its own simulated *system time* and (ii) makes a *virtual machine* available to the user.

The virtual machine may be a representative from a wide variety of machines, which may even be non-existent or not available at the moment. Machine parameters are read from a file when the program starts. These parameters may be changed dynamically during the program execution.

The virtual layer makes it possible to compare program runs on computers having different communication latency and computation speed (independent of actual load characteristics). Timing experiments using different load balancing strategies can be made easily and quickly and thus enable the determination of the optimum load balancing strategy for certain architectures[1].

[1] For short execution times stochastic effects may possibly overlap timing data due to com-

2.2 Using MISS-PVM

PVM 3 is a software system linking a network of Unix computers designed to mimic the existence of a single large (parallel) computer. It provides message passing and process control routines for tasks running on different computers. PVM 3 uses daemon processes on every host[2] to establish communication from one user process to another one. The user processes can send their messages only to the PVM daemon on their actual machine, the PVM daemon communicates with the PVM daemon of the target machine, and this PVM daemon delivers the message to the receiver. User programs are linked to the PVM 3 library, which contains interface routines to the `pvm daemon`.

Figure 2.1 shows the relationship between the parallel user program, PVM 3, and the native communication primitives available on the host computer. The user program calls PVM 3 subroutines to transfer messages between different processes, and to create and to terminate processes on various network nodes. The PVM 3 subroutines in turn perform their tasks by calling native communication primitives of the underlying computer system. In this way a user program can be run without modification on a variety of different computer systems, as the use of PVM 3 subroutines makes machine dependent, native communication primitives transparent.

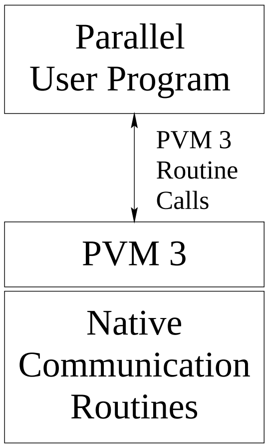

Figure 2.1: Position of PVM.

As a preparatory step in the development of the machine simulator CLUE a new level between the user program and PVM 3 is added: the *Virtual Layer for PVM 3* (see Figure 2.2). It uses only PVM 3 calls and contains no machine dependent routines. Thus the *Virtual Layer for PVM 3* is portable to a wide variety of machines (like PVM). This library also boasts other advantages by

puter timing routines that are too coarse grained.

[2]The PVM daemon is often abbreviated with `pvmd` or `pvmd3`.

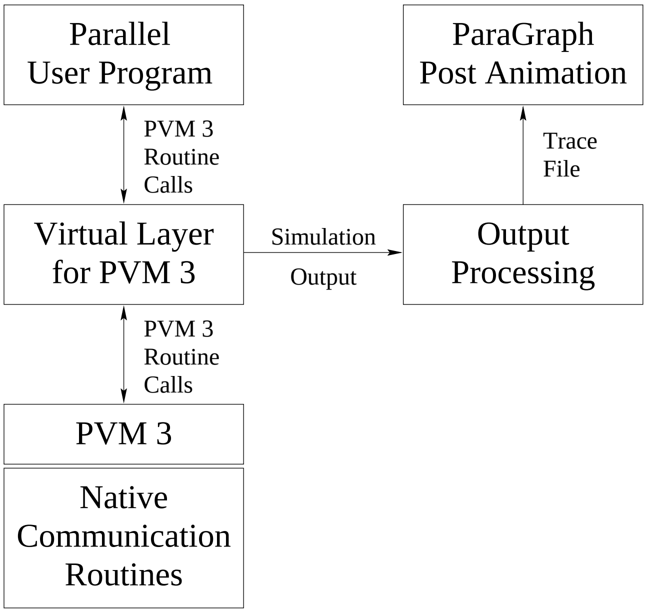

Figure 2.2: Position of the Virtual Layer for PVM.

providing virtual time, virtual machines, and output generation for graphical post mortem visualization. The user program as well as the PVM 3 level remain unchanged. The only difference is that an include file redirects PVM 3 routine calls. The virtual layer creates output which is post processed by simple programs and is then used as input for post mortem visualization using ParaGraph.

So there are three steps in every program run.

Execution of the Parallel Program. Two modifications have to be made within a program to be investigated. An include statement has to be changed and an additional library has to be linked to the program. Calls to a PVM 3 routine should take place at the beginning of every program being executed in parallel, so that correct timing is ensured. Standard timing routines should not be called, because system time routine calls do not return the simulated system time (with the exception of the `time()` call, which is also redirected to MISS-PVM). `pvmS` calls (see Section 2.4) can be added in order to use special properties of MISS-PVM, such as obtaining the virtual time directly.

Calling the Output Processing Program. This is done by calling a script file after the program execution to collect the output files of all tasks running in parallel and to prepare these output files to be read by ParaGraph. In order to write data to the output file the script file has to be provided with the name of its output file as a parameter. The script file then adds the extension `.trf` to this name.

Visualizing the Output Using ParaGraph. PARAGRAPH takes the full name of the output file (including the extension `.trf`) as parameter. Visualization takes place in PARAGRAPH via various animation windows, which are, to a great extent, self-explanatory (see Heath [12], Tomas, Ueberhuber [16]).

2.3 Communication Libraries and MISS-PVM

Once the high level library routines (based on PVM) have been recompiled, no further modifications are needed. The user program is linked with the new library. The result is a program that performs the same task as before, except that it writes an output file and can be simulated on virtual machines.

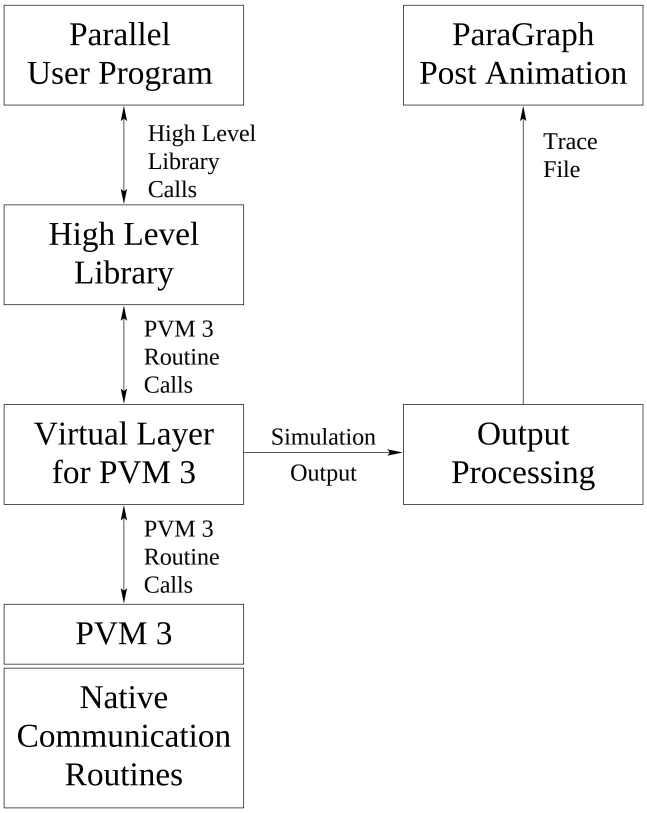

Figure 2.3: High level communication library, e. g., the basic linear algebra communication subroutines (BLACS), used in conjunction with PVM 3.

Figure 2.3 shows that a new level has been added to the structure depicted in Figure 2.2. A high level library is situated between the user program and the *Virtual Layer for PVM 3.* The user program remains unchanged when the high level library is recompiled, and the virtual layer works with the routines in the same way it works with normal user programs.

When working at a higher level of abstraction than is provided by PVM 3 not every ParaGraph visualization is easy to understand. Post mortem visualization should be limited to general information windows showing, for instance, the communication/computation ratio.

2.4 Virtual Time

The *Virtual Layer for PVM 3* uses an internal time based on three components.

Computation Time which is obtained by (i) measuring the CPU time consumed by the executed user programs and (ii) modifying by performance factors.

Communication Time which is calculated using the parameters of the virtual machine.

Waiting Time obtained in parallel execution.

The sum of these time components is the "virtual time" of each user process.

The internal timing granularity of the virtual time is normally between 100 microseconds and 1 microsecond. Time output is always measured in microseconds. In low resolution timing results, additional digits have to be inserted, because in ParaGraph the input is sorted according to time. No two time stamps are allowed to be the same for each task.

2.5 Virtual Machines

Virtual Machines are defined in the file `VMConfig`. The format of this file is as follows: In the first line are the parameters of the machine used for the master program and as default for all programs started without an explicit machine and host type given. In the other lines comments (beginning with the symbol `#`), or machine or host type specifications can be found.

Parameters found on specification lines are:

Name of the Machine or Host Type. This name is used in `pvm_spawn`. If the following parameter is 0, the machine is assumed to be "real", and the program is started on this machine. Otherwise the machine has a "virtual" name, and PVM 3 is asked to look for a suitable machine.

Performance Factor. This is a floating-point multiplier for computation time. If this parameter is 0, the machine name is sent to PVM 3 and the computation timing results are not changed.

Initialization Delay. This is the time needed for `pvm_spawn` to be called.

Send Delay. This is the time used for sending a message using `pvm_send` or `pvm_mcast`. This time contains packing the message, resolving the address of the host and starting the transmission (as far as the sending process is involved). This time is independent of the message length.

Receive Delay. This is the time used in calling the receive routines `pvm_rcv`, `pvm_nrecv` and `pvm_probe`. This time is always the same whether these routines succeed or fail.

Transmission Delay. This is the time used to transfer a message which is independent of the message length.

Transmission Proportional Delay. This is the time used for the transfer of a message which depends on the message length. This time is measured in 100 microseconds per Kbyte.

These parameters are used in the performance modeling file `delay.c`. If the actual performance model turns out not to be exact enough, it can easily be changed by modifying this file.

2.6 Transparency

In the development of MISS-PVM it was crucial to maintain all PVM 3 functions. When PVM 3 is used in conjunction with the virtual layer, virtual timing routines and virtual machine properties are added; otherwise there are no other modifications (cf. Sections 2.4, 2.5, 3).

Virtual timing routines should replace system timing routines. This is because system time calls can only return real time or process time, which is inappropriate for the user program's time. Virtual time is calculated from process time, but it is modified by adding certain delays from inter-process communications and by subtracting overhead time that results from simulation using the virtual layer.

Using the virtual layer entails the simulation of virtual machines, which leads to a situation in which it is not clear which computer actually executes user processes. So user programs should not try to establish perfect communication patterns by determining their actual host.

This unpredictability in determining the actual host results in the following situation: If programs are started on a cluster of inhomogeneous computers, time measurements are not reliable, since some machines will execute a certain code faster than others. There are two solutions:

- The first one is to run all program instances on the same type of machines (a homogeneous cluster) or on a single machine. MISS-PVM spawns a new task on a machine that was selected by PVM 3 if either no machine name is given in the call to `pvm_spawn` or if the machine performance factor in the file VMConfig has a value different from 0.

- The second (and more difficult) solution is to spawn processes on different machines which are described in the virtual machine configuration file `VMConfig` (see Section 2.5). Spawning a new task onto a virtual host is done by using the virtual host name (the first parameter of a line in the file `VMConfig`) as host name and by setting the flag parameter to `PvmTaskHost` (1) or `PvmTaskArch` (2).

Chapter 3

Restrictions of MISS-PVM

When using MISS-PVM, the simulation designer has to be aware of certain restrictions that limit the applicability of MISS-PVM. Neglecting these restriction may lead to unpredictable and inaccurate results.

3.1 Restrictions due to Execution Time

For large programs (especially those which perform compute intensive tasks) execution time can become a prohibitive factor in simulation. It is advisable to begin with small problem dimensions (like the dimension of a linear system or the number of iterations). Not only the execution time for a program slows down computation but also the amount of memory that is used. If there are more parallel running processes than compute nodes, each process running in parallel on a single node will use much memory. This can lead to performance degradation if the machine starts to swap memory to the hard disk.

Many difficulties that arise during the construction of parallel programs do not need large scale problems to be eliminated, they can also be solved using small scale problems. ParaGraph is not easy to use for input files that are much bigger than one Mbyte. This is due to one of two reasons. The first is that too much information is given to the user in too short a time, which leads to an animation that does not give enough detailed information. The second is that it can take too long to get to that part of the simulation which is of interest.

3.2 Internal Restrictions

When using MISS-PVM the user has to deal with certain internal restrictions. The use of MISS-PVM will not be prevented by any of these restrictions, but some differences to the use of PVM 3 without MISS-PVM can result. Normally the user will not notice these differences. However, unexpected results may occur (without explicit error messages) due to one of the following restrictions. Error messages may be hard to find, because the output of child tasks does not always go to the user terminal. In these cases the user has to look for messages in the file `/tmp/pvml.`*uid* (*uid* is the user ID).

- The master task must not be called from the PVM 3 command line. It would wait forever for initialization data, because the `pvm_parent()` routine behaves different in this case.

- `pvm_mytid` (or any other PVM 3 routine) has to be called as soon as possible after the task starts (in order to generate correct timing measurements). After the master task has sent a message to one of its children, the children's clocks are automatically synchronized with the master clock.

- `int` must be 32 bit wide in order to ensure correct timing. Program runs longer than $200,000$ seconds produce a timer overflow.

- Group routines are not supported by the virtual layer.

- All hosts in the cluster must have the same performance characteristics in order to produce reliable computation timing output.

- The length of messages sent is not reported (because this is not supported by PVM 3). It is reported, however, after a receive statement.

- Empty `pvm_nrecv()` loops may run a very long time, because every call of this routine produces a line (or two) in the output file. While these lines are being written, the simulation time is advanced very slowly.

3.3 Restrictions in the Use of Specific Routines

There are complications involved with using certain PVM 3 routines adapted to the virtual layer. These routines are modified, but do not return error conditions if there is an error. The complete list of these routines is as follows:

- `pvm_kill` — in situations where error conditions are returned, it is possible that return values are incorrect. The reason for this is that it is not possible for one task to kill another in the virtual layer. The task to be killed might not have an advanced enough virtual time to be killed. What happens instead is that a message is sent to the task to be killed to "commit suicide" at a given time. The "killer task" does not wait for the other to "commit suicide". This is why the "killer task" does not return the correct value.

- `pvm_tasks` — the virtual host computer is not reported correctly.

- `pvm_mstat` — the status of virtual hosts is not reported.

- `pvm_config` — could give more information about all virtual hosts, but it is not supported yet.

- `pvm_recvf` — due to the internal restructuring of receiving statements and the hope that nobody will ever need this routine when using MISS-PVM, the virtual layer does not support it correctly.

Chapter 4

Numerical Experiments

This chapter is split into three sections: the first briefly describes the simulation and shows how to get performance data on existing clusters, the second validates these results by comparison of simulated and measured programs.

4.1 Experimental Environment

Up to now, two Beowulf clusters have been examined.
The **Vienna Cluster** has the following configuration (see Section 1.1):

- Suse Linux 6.0
- 10 processors in 5 nodes
- 2 $\times$ 350 MHz Pentium II per node
- Switched Fast Ethernet (100 Mbit/s)

PCs of this cluster have also been used to run simulations. In this case a computational factor of 1.0 has been assumed.
The second cluster is being situated at RWTH Aachen. The **Aachen Cluster** has the following configuration (see Section 1.1):

- Siemens hpcLine (Linux)
- 32 processors in 16 nodes
- 2 $\times$ 400 MHz Pentium II per node
- Scali SCI network

The topology of this SCI cluster is a 2-dimensional torus.

4.2 Measurement and Modeling

On both clusters, measurements have been carried out to obtain the parameters for sending messages. A C program was written, consisting of a master and a slave part. First, the master starts the slave by calling `pvm_spawn()`. Then, the difference of the host clocks is measured by sending `udp` packets. Then, the server opens a socket and waits for `udp` packets from the client. The master eventually starts sending single packets, recording its host time for each sent packet. Upon receiving the packet, the slave gets the current time and sends it back to the master. The master records the time for sending the `udp` packet to the slave and back (ping-pong time), together with the time difference of the slave. After N ping-pongs, the host time difference is then measured by using the host time difference with the smallest ping-pong time (best estimate).

After estimating the clock difference, the master starts sending messages via PVM, packing its time into the message. The slave extracts this time and can then use its own clock and the measured host difference to obtain the transmission estimate. The recorded estimate is then computed by taking the mean transmission time of all messages of the same size.

This procedure has been applied to measure the time messages need when sent from one host to the same host, and from one host to a different host over the network. This time is split into two parts:

- Send time: The time that the sender spends in the send call.
- Transmission time: The difference between the send time and the time that the message needs to be received by the receiver.

Vienna Cluster. Figure 4.1 shows the send and transmission times observed on the Vienna cluster. Both sender and receiver run on the same node, thus the message is not sent over the network. Also, the piecewise linear model used for the simulation in shown as well.

Figure 4.2 shows the piecewise linear model that is used to simulate messages sent from one node to another.

Additionally, for the case of sending messages from one sender to several receivers at the same time, contention has been observed that increases both the send and transmission time. Figure 4.3 shows the observed contention factor for the send time. Both sender and receiver are on the same node. This is important for those cases, where all tasks synchronize, and one of the tasks then sends out messages to one or several others to redistribute work or results.

Aachen Cluster. The computational factor of nodes of the Aachen cluster has been measured to be 0.91 relative to the Vienna cluster, where simulation runs have been carried out. Additionally, the SCI network was only available for the MPI version of BLACS. Thus, the communication parameters and the performance of the real runs were collected for the MPI version of BLACS, whereas the simulation runs were still carried out on the Vienna cluster using the PVM version of BLACS.

Figure 4.4 shows the send and transmission models for the Aachen cluster. This model has been observed independently of the location of sender and receiver. It can be seen that the send time (the time that the sender spends in

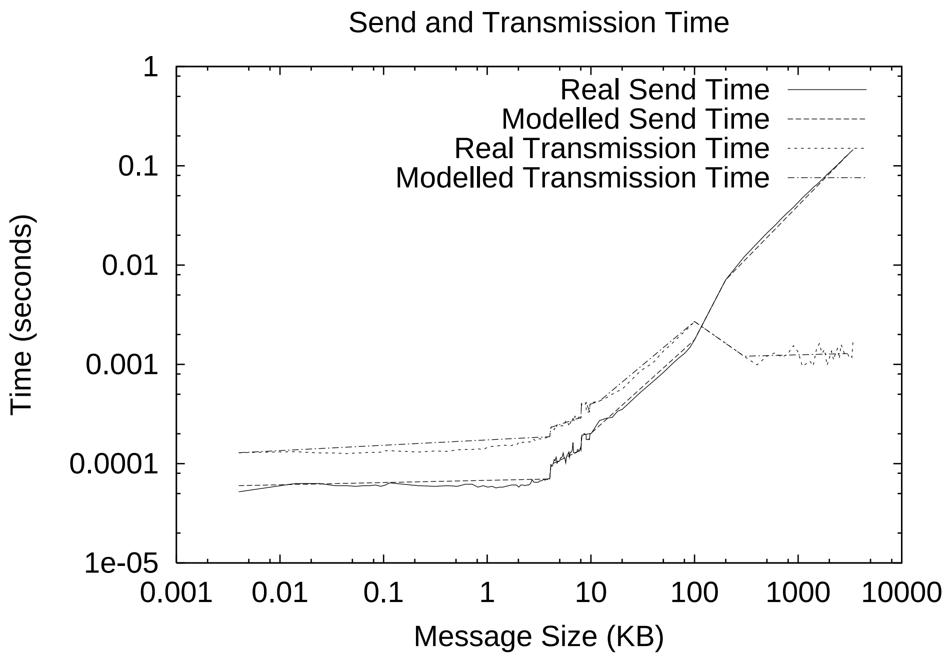

Figure 4.1: Send and transmission time for the Vienna cluster. Sender and receiver are on the same node.

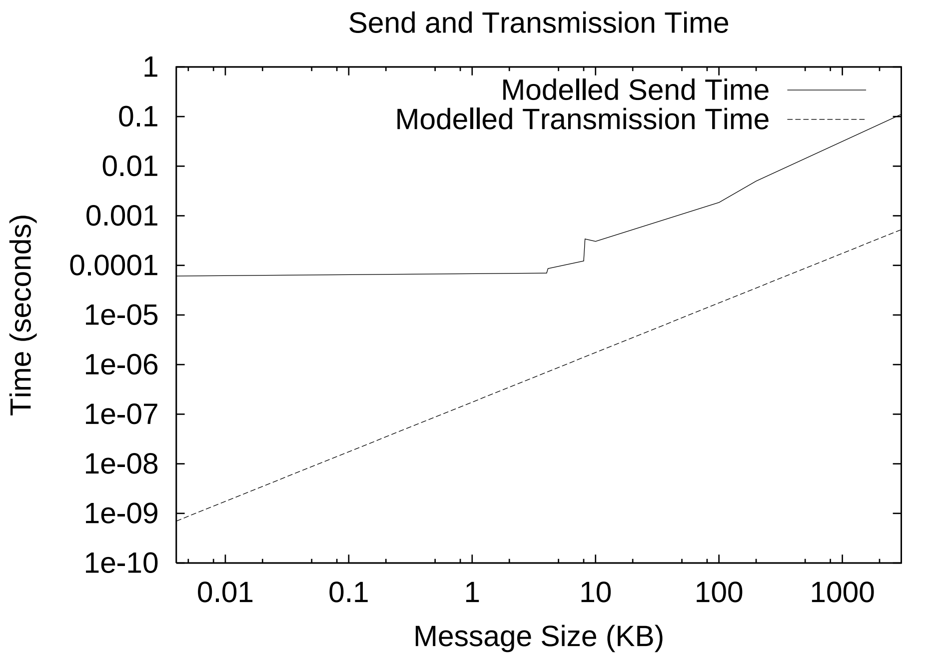

Figure 4.2: Send and transmission models for the Vienna cluster. Sender and receiver are on different nodes.

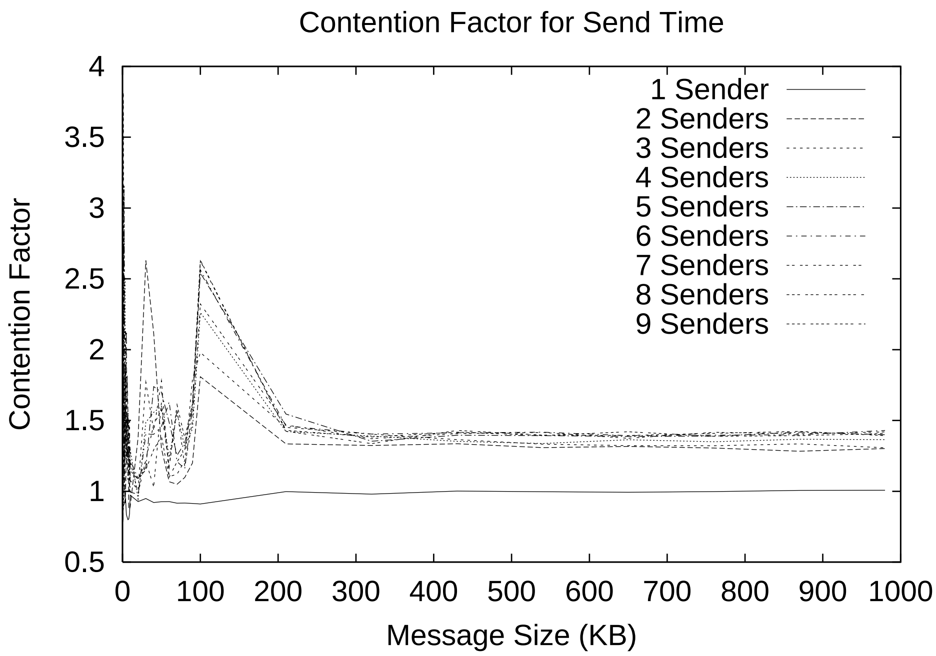

Figure 4.3: Contention factors for the send time on the Vienna cluster. Sender and receiver are on the same node.

the send call) is much smaller than the overall time (send time plus transmission time). One consequence is that no contention has been measured for the Aachen cluster.

4.3 Simulation Results

In order to obtain generally applicable results, three important numerical algorithms (see Section 1.3) have been chosen to be simulated on the two clusters.

1. Parallel Cholesky factorization (ScaLapack).
2. Parallel LU factorization (ScaLapack).
3. Parallel matrix-matrix multiplication (PBlas).

2000×2000 matrices were chosen. This matrix size represents a trade-off between computational complexity needed to obtain meaningful results, and the main memory requirements of the simulating node.

For the real runs, the PVM version of ScaLapack and PBlas were used (on the Vienna cluster). For the simulation runs, a copy of the Blacs library was made, and in the file `Bconfig.h`, the entry `#include <pvm3.h>` was changed to `#include <pvm3V.h>`. Additionally, the MISS-PVM library was linked to the

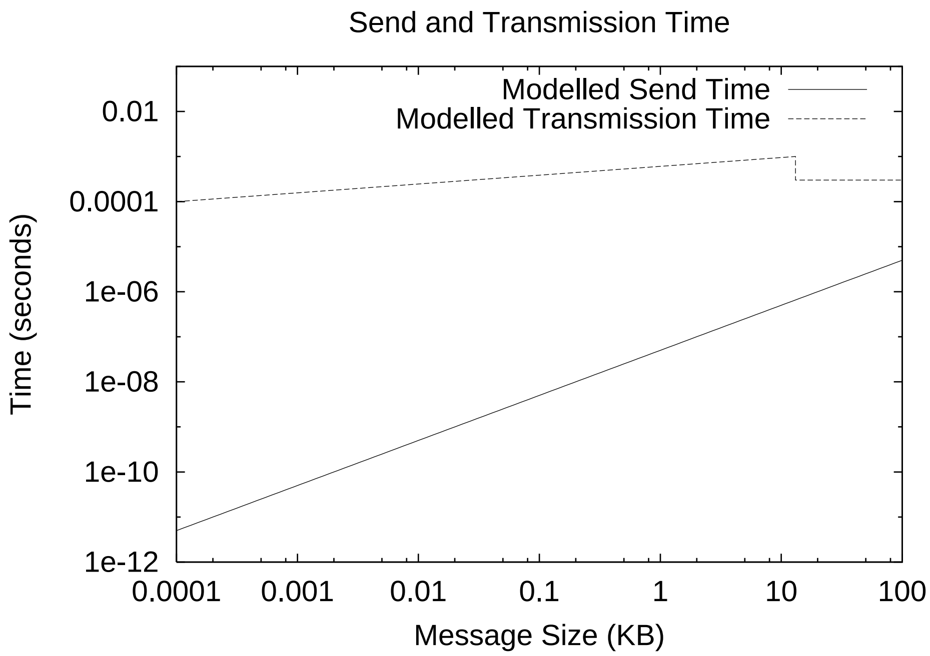

Figure 4.4: Send and transmission models for the Aachen cluster.

executables. All simulation runs were then carried out on workstations of the Vienna cluster. Each simulation run was carried out on one workstation only.

All executables report their result in terms of CPU time and wall time. As the CPU time is meaningless for the simulation, the reported wall time has been collected for each run.

The simulation runs should answer the following questions:

1. Do the real observations and the simulated runs have the same qualitative properties?

2. Do the real observations and the simulated runs have the same quantitative properties?

3. Can the simulation results be used to evaluate the performance of workstations clusters a-priori?

In the following figures, the observed and simulated wall times are plotted against the processor grid used. Such a grid or 2-dimensional mesh is always assumed to define the topology of the parallel computer, even if in reality this is a workstation cluster connected over a bus, star or ring topology. Each processor is assigned to a certain place in the virtual mesh topology. Basically, an $N \times M$ grid means that $N \times M$ processors were used to compute the task. The relation of N to M defines the communication pattern used, yielding different speed-ups as the below results show.

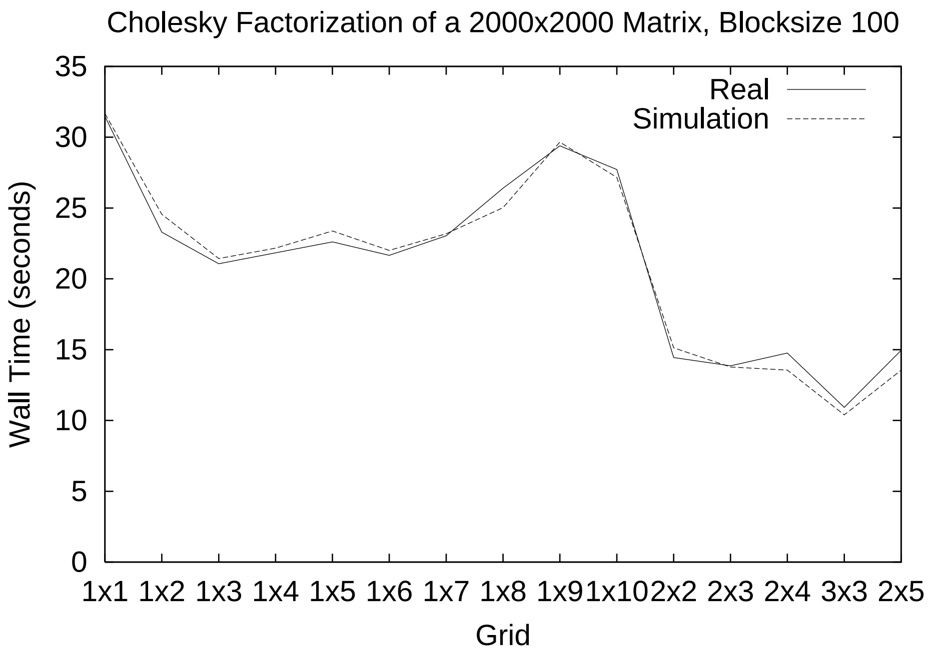

Figure 4.5: Cholesky factorization on the Vienna cluster. Blocksize is set to 100.

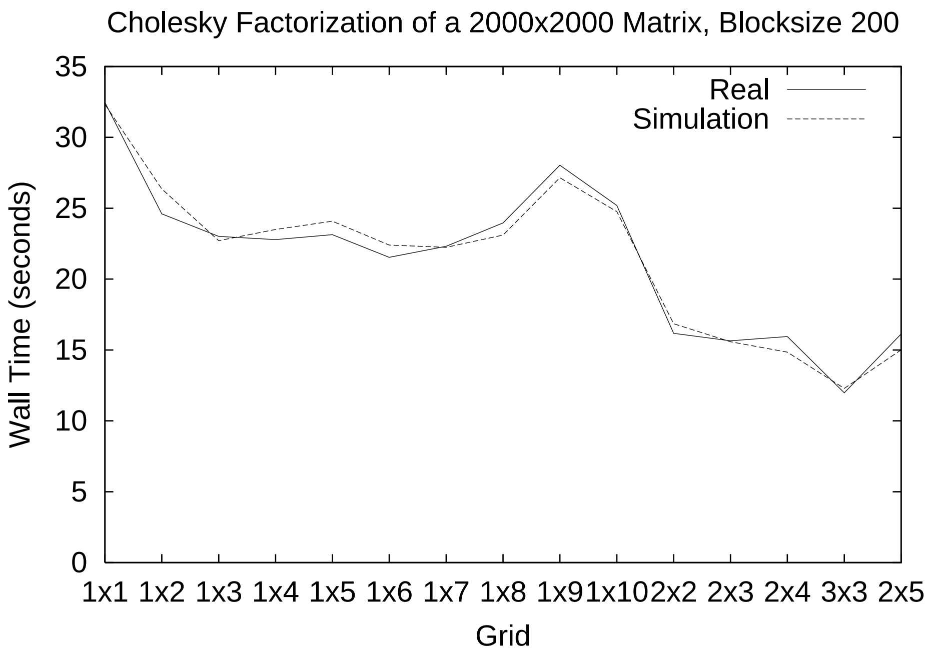

Figure 4.6: Cholesky factorization on the Vienna cluster. Blocksize is set to 200.

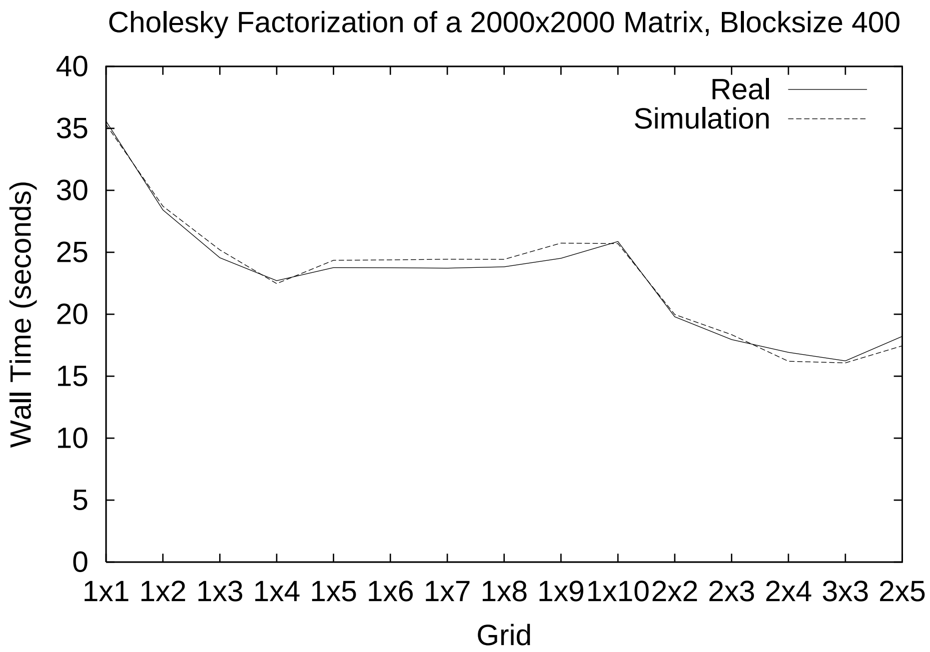

Figure 4.7: Cholesky factorization on the Vienna cluster. Blocksize is set to 400.

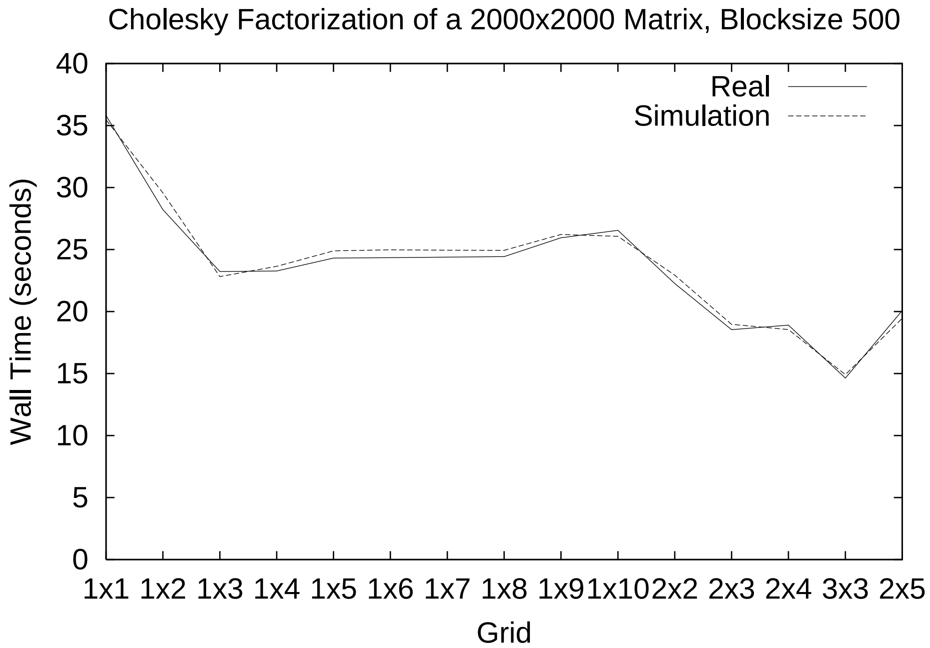

Figure 4.8: Cholesky factorization on the Vienna cluster. Blocksize is set to 500.

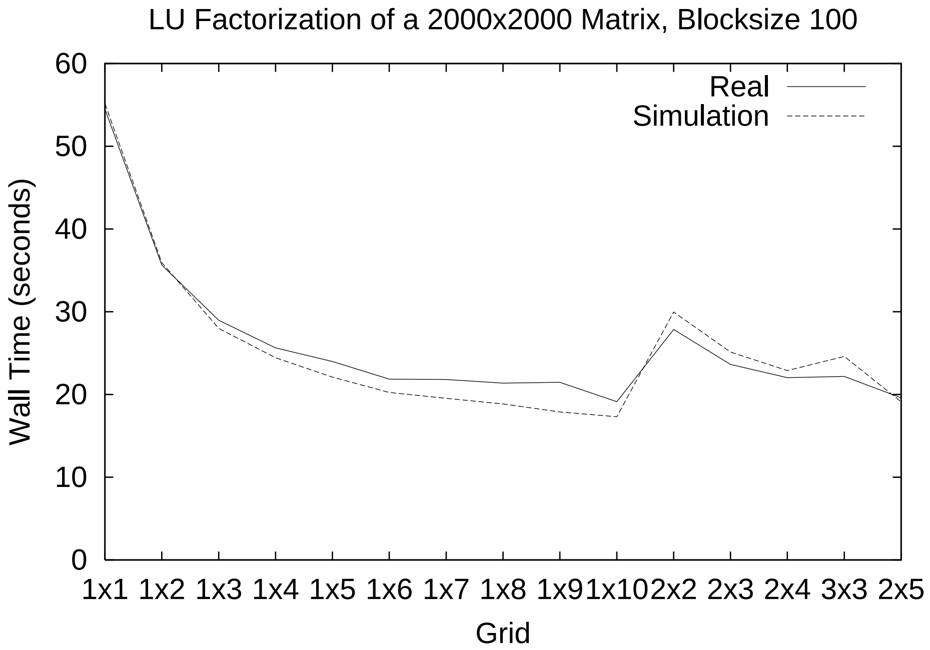

Figure 4.9: LU factorization on the Vienna cluster. Blocksize is set to 100.

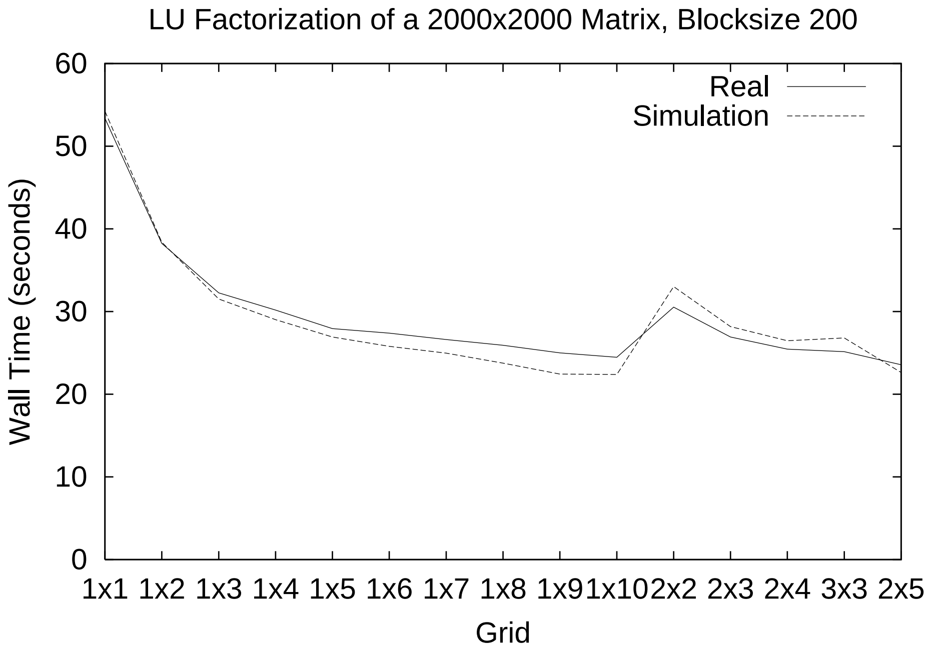

Figure 4.10: LU factorization on the Vienna cluster. Blocksize is set to 200.

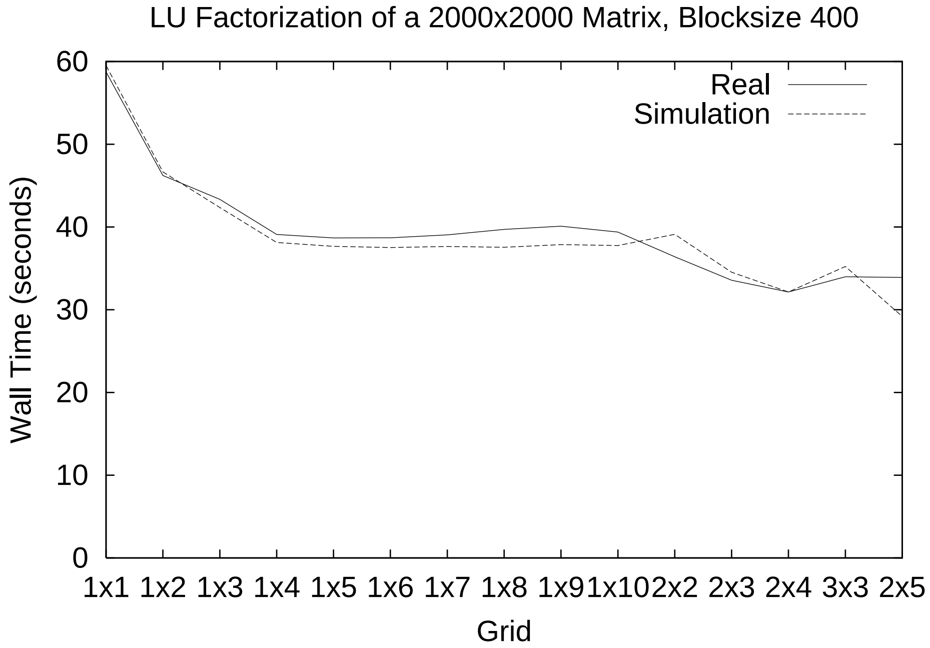

Figure 4.11: LU factorization on the Vienna cluster. Blocksize is set to 400.

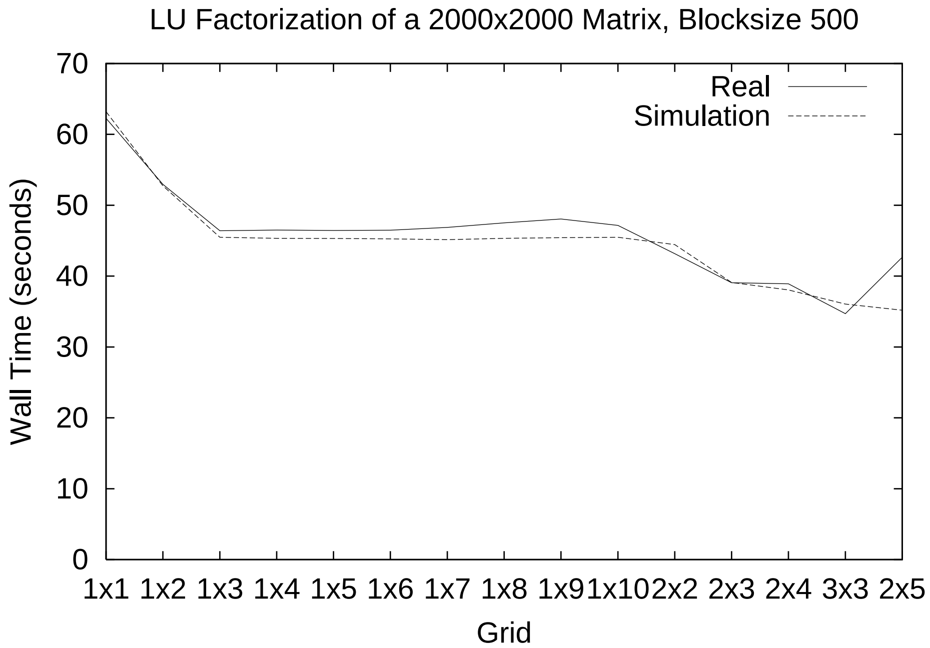

Figure 4.12: LU factorization on the Vienna cluster. Blocksize is set to 500.

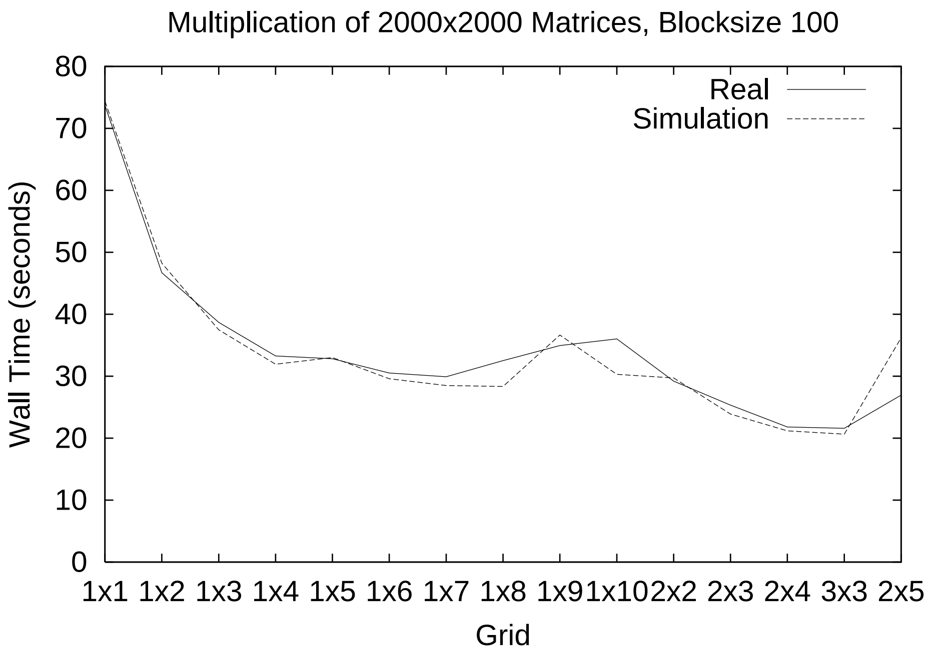

Figure 4.13: Matrix-matrix multiplication on the Vienna cluster. Blocksize is set to 100.

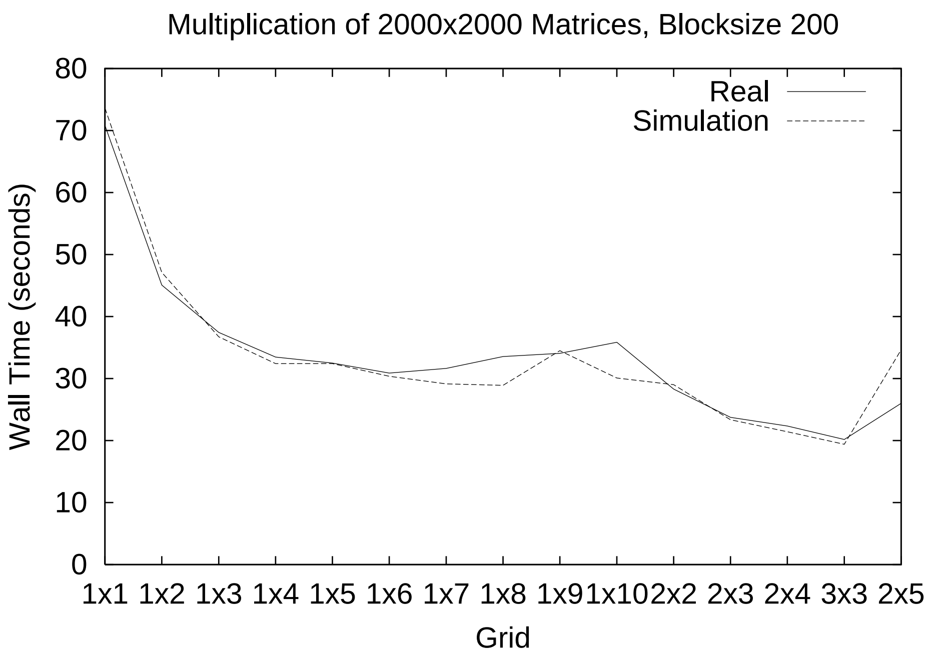

Figure 4.14: Matrix-matrix multiplication on the Vienna cluster. Blocksize is set to 200.

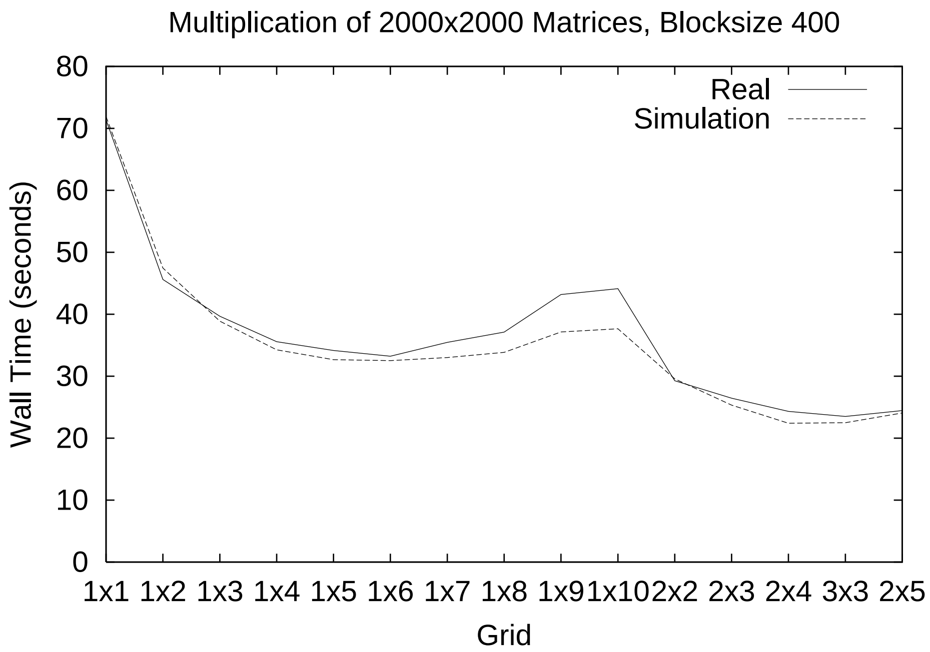

Figure 4.15: Matrix-matrix multiplication on the Vienna cluster. Blocksize is set to 400.

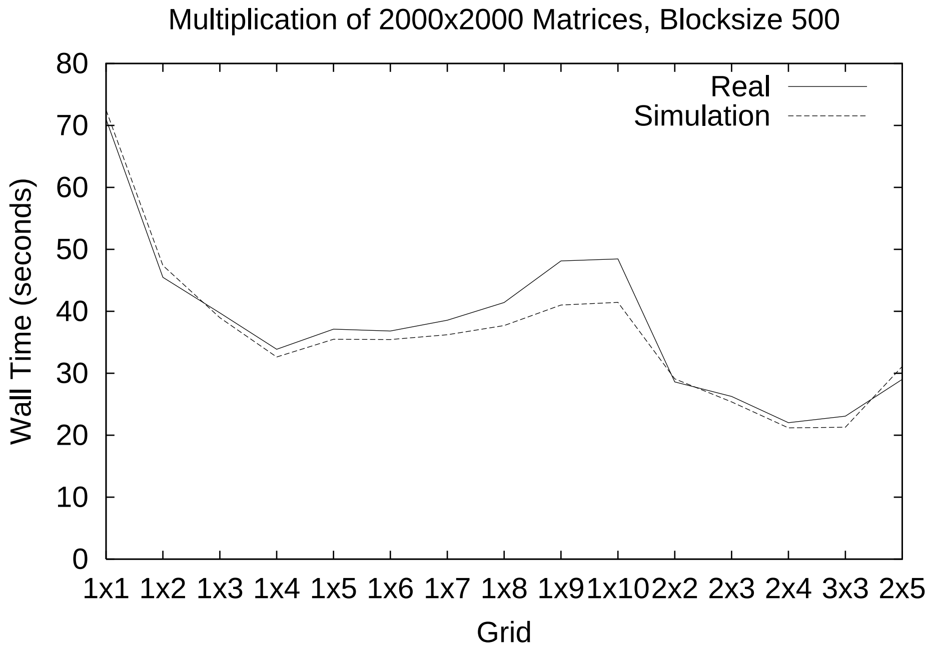

Figure 4.16: Matrix-matrix multiplication on the Vienna cluster. Blocksize is set to 500.

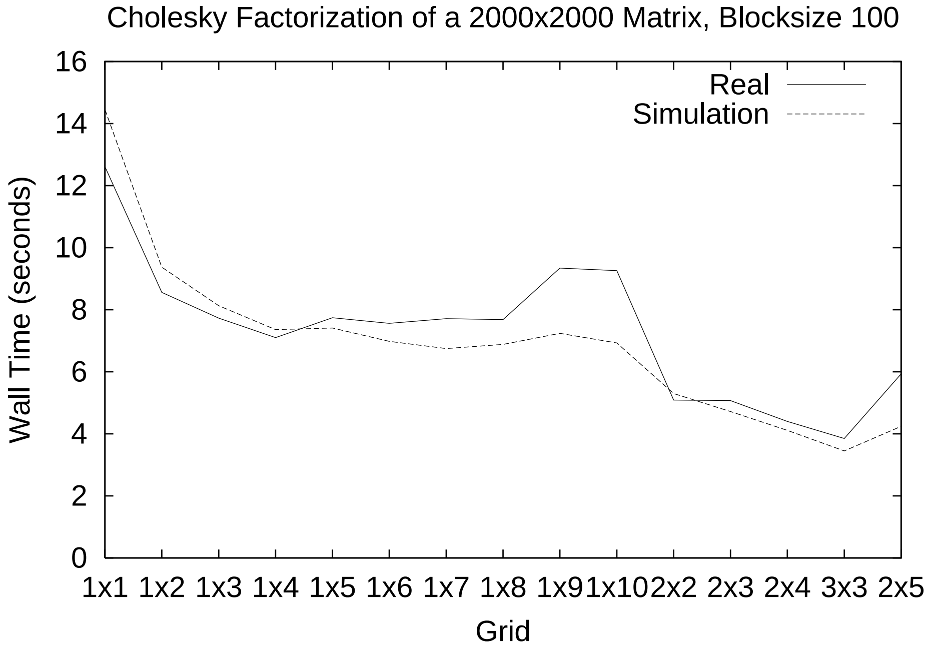

Figure 4.17: Cholesky factorization on the Aachen cluster. Blocksize is set to 100.

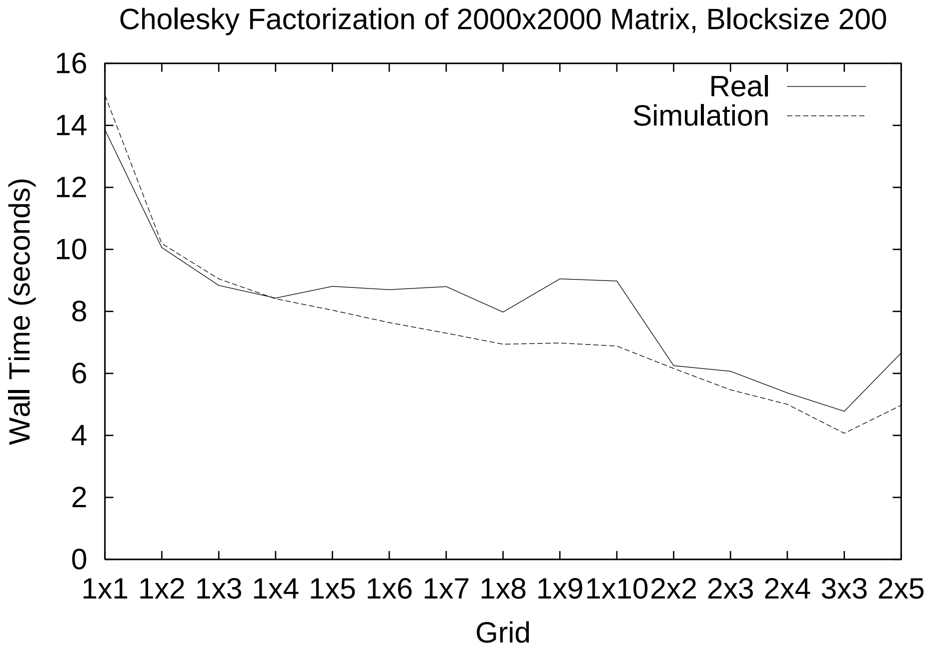

Figure 4.18: Cholesky factorization on the Aachen cluster. Blocksize is set to 200.

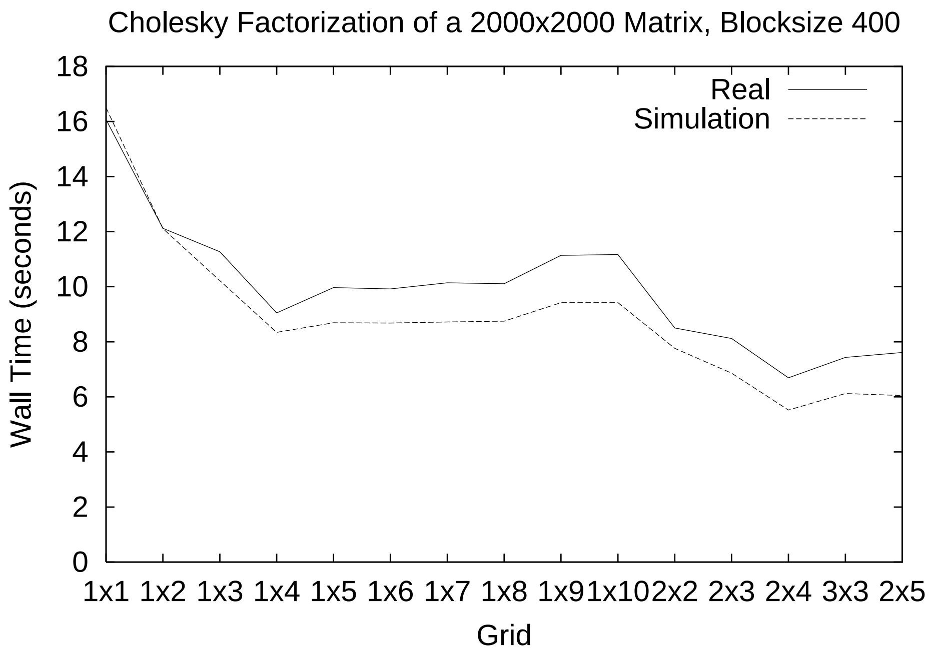

Figure 4.19: Cholesky factorization on the Aachen cluster. Blocksize is set to 400.

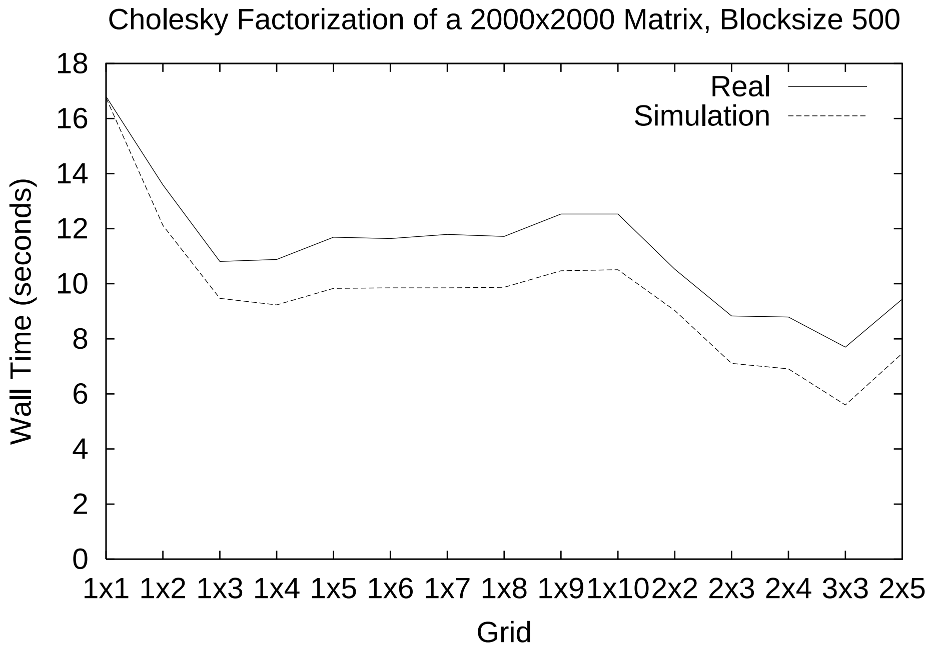

Figure 4.20: Cholesky factorization on the Aachen cluster. Blocksize is set to 500.

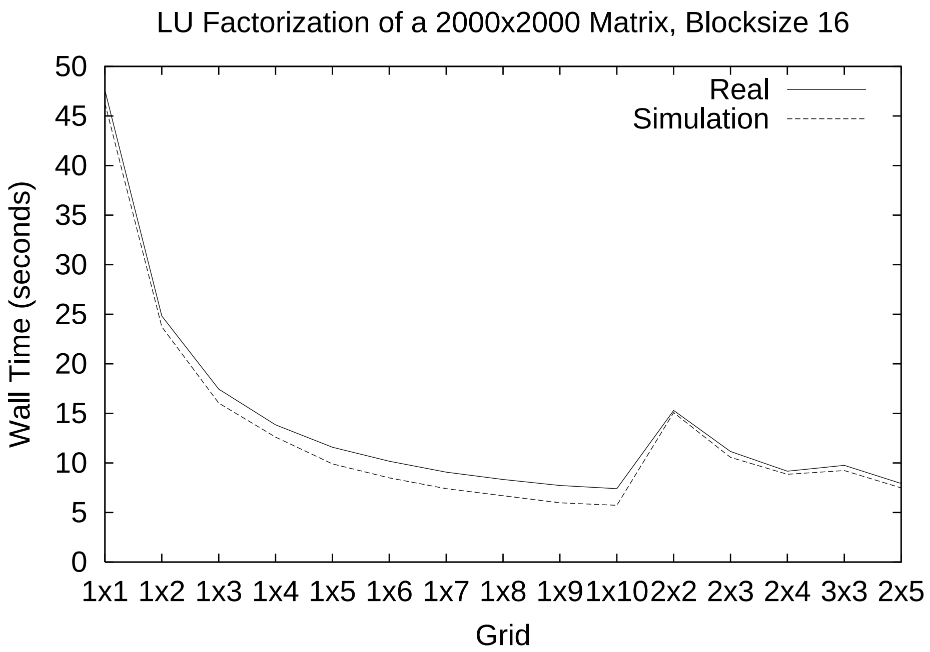

Figure 4.21: LU factorization on the Aachen cluster. Blocksize is set to 16.

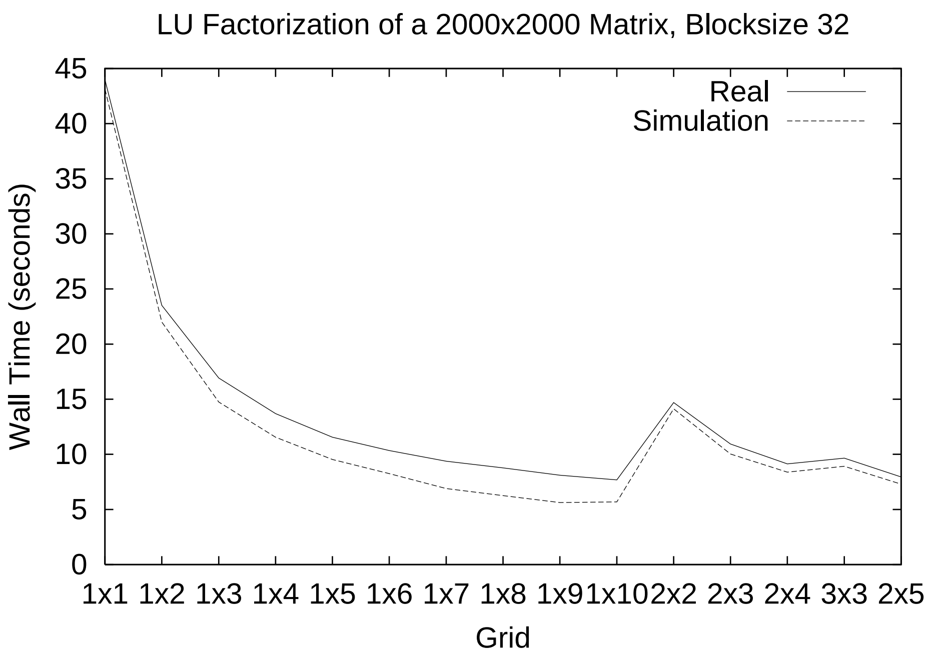

Figure 4.22: LU factorization on the Aachen cluster. Blocksize is set to 32.

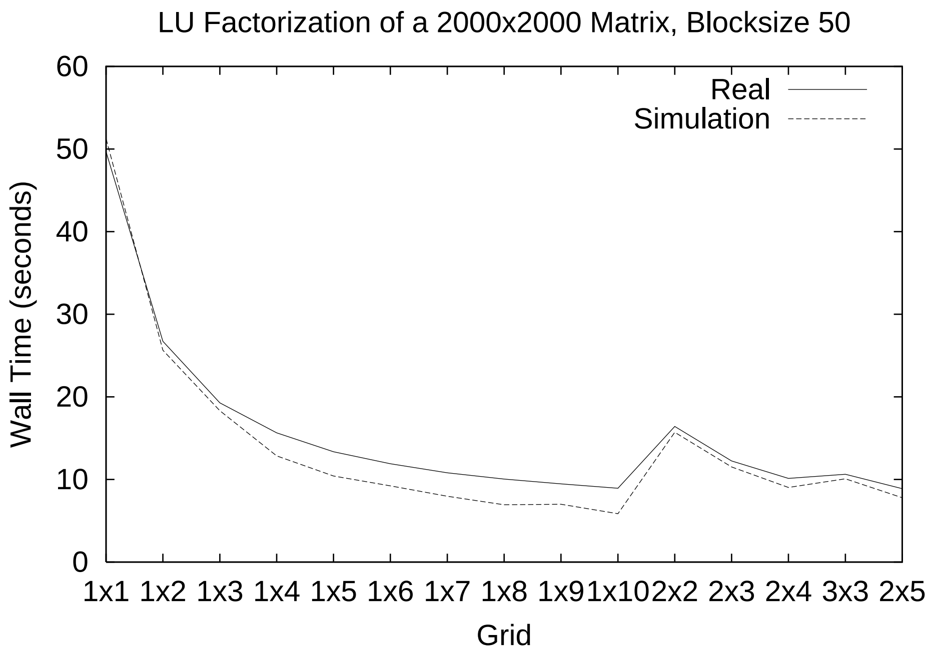

Figure 4.23: LU factorization on the Aachen cluster. Blocksize is set to 50.

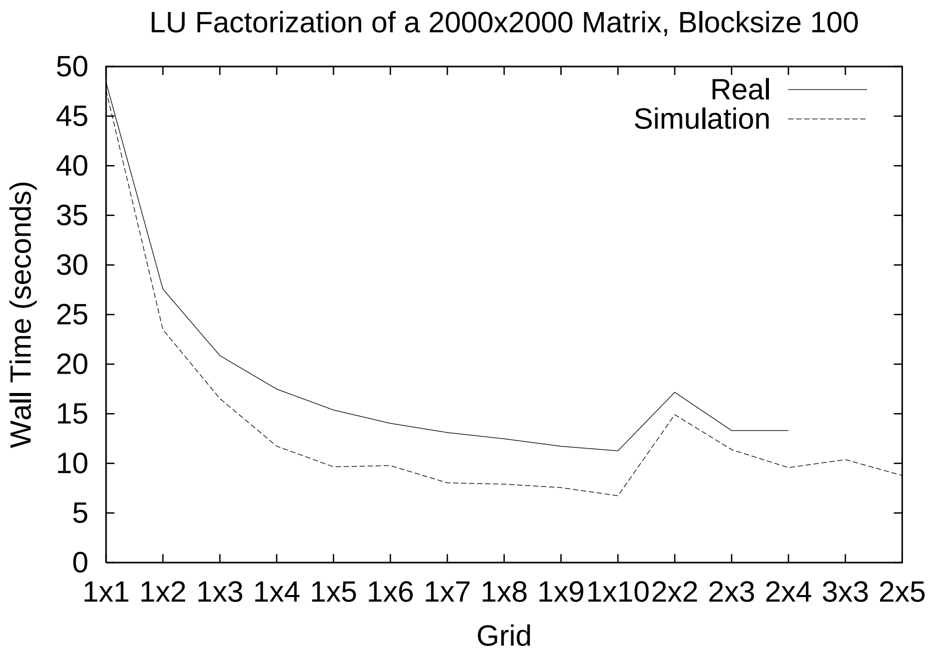

Figure 4.24: LU factorization on the Aachen cluster. Blocksize is set to 100.

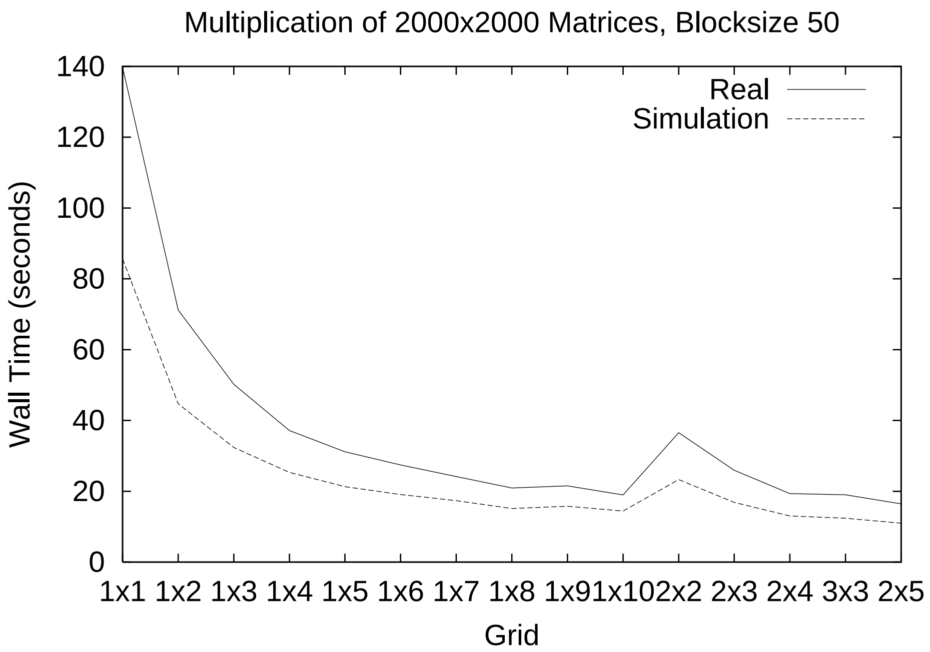

Figure 4.25: Matrix-matrix multiplication on the Aachen cluster. Blocksize is set to 50.

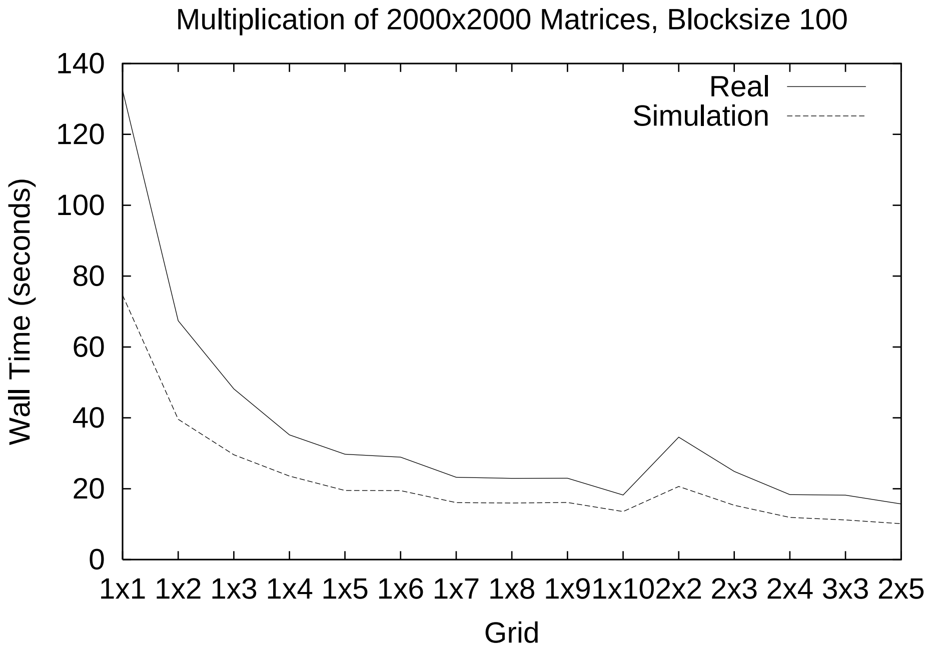

Figure 4.26: Matrix-matrix multiplication on the Aachen cluster. Blocksize is set to 100.

Chapter 5

Conclusion

In this report a new simulation tool called CLUE was described. This tool is meant to investigate the performance behavior of clusters of SMPs.

Simulation results obtained for a PC cluster with slow communication (using Fast Ethernet) are very accurate. Both qualitative and quantitative performance behavior of this Beowulf cluster has been captured by the simulation highly satisfactory. Inaccuracies only occur for some runs of matrix-matrix multiplication, where some contention is not reflected by the simulation.

In contrast to simulating the performance of the PVM versions of ScaLAPACK and PBLAS by using the same PVM code, simulating their MPI versions by using the PVM versions on a different type of node is far more complicated. Still, performance diagrams indicate that the qualitative behavior of the parallel programs is accurately simulated, while the quantitative results are sometimes a little misleading.

It may thus be concluded, that performance comparisons between different workstation clusters are possible, though experiments must be carefully designed and interpreted. The qualitative behavior of parallel programs running on a workstation cluster though can be simulated accurately, independent of the use of PVM or MPI.

It is thus possible to analyze the behavior of parallel programs and predict their performance, depending on cluster parameters. Simulation results can be used to investigate the influence of different parameters of the simulated workstation or PC cluster, in order to plan new hardware configurations or make an educated choice between several alternatives.

Chapter 6

Future Work

Future work will include the development of improved network models to simulate various topologies. Also, classical network simulators like OPNET or NS will be used to develop new ways of estimating simulation parameters for workstation or PC cluster configurations that are not (yet) available.

Furthermore, the performance of several other clusters will be investigated in order to improve the simulation methodology.

Appendix A

Blocking for Parallelism

The layout of an application's data within the hierarchical memory of a concurrent computer is a critical factor determining the performance and scalability of parallel code. On single processor and parallel computers it is important to make efficient use of the hierarchical memory by maximizing data reuse. For example, on a cache-based computer, block-oriented matrix-matrix operations are to be implemented by using the Level 3 BLAS whenever possible.

Localized data access generally results in increasing the ratio of floating-point operations to memory references and enables data reuse as much as possible while data is stored in the highest levels of the memory hierarchy (like vector registers or high-speed cache).

An analogous approach has to be followed in the design of programs for (parallel) distributed-memory machines. By using block-partitioned algorithms a reduction of data transfers is achieved, thereby reducing the overall latency.

Block-Cyclic Data Distribution. The choice of an appropriate data distribution heavily depends on the computational flow in the algorithm. Dense matrix computations, which are used as examples in this report, feature a large amount of parallelism, so that many different distribution schemes have the potential for achieving high performance.

The two main issues in choosing a certain data layout for dense matrix computation are

- load balancing, i. e., splitting the work evenly among the processors throughout the algorithm, and
- use of Level 3 BLAS in matrix computations on single processors, to efficiently deal with the memory hierarchy of each processor.

Many algorithms in linear algebra (LU decomposition, Cholesky factorization, tridiagonalization, etc.) work on successively smaller square submatrices of the original matrix, ending up in the very first (a_{11}) or very last (a_{nn}) matrix element.

In the following processes are numbered from 0 to $P-1$, and matrix columns (or rows) from 1 to n. Figures A.1 and A.2 show data layouts to be considered. In all cases, each submatrix is labeled with the number of the process (from 0 to 3) that owns it. Process 0 owns the shaded submatrices.

The layout illustrated on the left of Figure A.1 is the *one-dimensional block column distribution*. This data distribution assigns blocks of contiguous columns

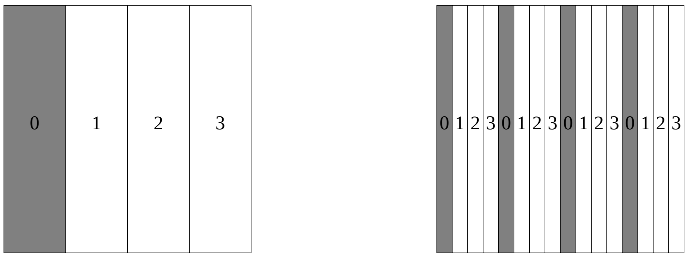

Figure A.1: One-dimensional block column and cyclic column distribution.

to successive processes. Each process receives only one block of columns of the matrix. Column k is owned by process $\lfloor k/t_c \rfloor$ where $t_c = \lceil n/P \rceil$ is the maximum number of columns stored per process. In the figure, $n = 16$ and $P = 4$. This layout does not permit good load balancing in linear algebra algorithms because as soon as the first t_c columns are complete, process 0 is idle for the rest of the computation. The transpose of this layout, the *one-dimensional block row distribution*, has a similar shortcoming.

The layout illustrated on the right of Figure A.1 is the *one-dimensional cyclic column distribution.* It is assigning column k to process $[(k-1) \bmod P]$. In the figure, $n = 16$ and $P = 4$. Using this layout, each process owns approximately $(1/P)$th of the square southeast corner of the matrix, so the load is reasonably balanced. However, since single columns (rather than blocks) are stored, Level 2 or Level 3 BLAS cannot be used (concerning exceptions see Hendrickson et. al [13]). The transpose of this layout, the *one-dimensional cyclic row distribution*, has a similar drawback.

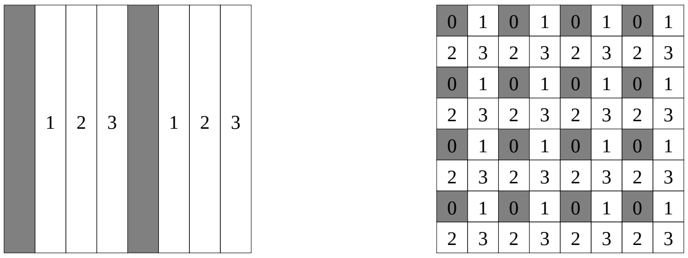

Figure A.2: One- and two-dimensional block cyclic distribution.

The third layout, shown on the left of Figure A.2, is the *one-dimensional block-cyclic column distribution.* It is a compromise between the two schemes of Figure A.1. A block size n_b is chosen, the columns are divided into groups of size n_b, and these groups are distributed in a cyclic manner. This means column k is owned by process $\lfloor (k-1)/n_b \rfloor \bmod P$. In fact, this layout includes the first two distributions as special cases, using parameters $n_b = t_c = \lceil n/P \rceil$ and $n_b = 1$, respectively. In Figure A.2, $n = 16$, $P = 4$ and $n_b = 2$. For $n_b > 1$, this layout results in a slightly worse load balancing than the one-dimensional cyclic column distribution. However, Level 2 and Level 3 BLAS can be used to carry out local computations. For $n_b < t_c$, it results in a better load balancing than the one-dimensional block column distribution. The BLAS can be called only on smaller subproblems. Hence, it takes less advantage of the local memory hierarchy. Moreover, this layout has the disadvantage that computations performed on one column block only (e. g., local factorization) will take place only in one process, thereby representing a serial bottleneck.

This serial bottleneck is eased by the fourth layout shown on the right of Figure A.2, the *two-dimensional block-cyclic distribution*. Here, the P processes are arranged in a $P_r \times P_c$ rectangular array, indexed in a two-dimensional fashion by (p_r, p_c), with $0 \leq p_r < P_r$ and $0 \leq p_c < P_c$. All the processes (p_r, p_c) with a fixed p_c are referred to as process column p_c. All the processes (p_r, p_c) with a fixed p_r are referred to as process row p_r. Thus, this layout includes all the previous layouts and their transposed counterparts, as special cases. In Figure A.2, $n = 16$, $P = 4$, $P_r = P_c = 2$, and $m_b = n_b = 2$. This layout permits P_c-fold parallelism in any column, and enables calls to the Level 2 and Level 3 BLAS on local subarrays. Finally, this layout also features good scalability properties.

The two-dimensional block-cyclic distribution scheme is the data layout used in the SCALAPACK library for dense matrix computations.

Bibliography

[1] E. Anderson et al., LAPACK *Users' Guide, 3rd ed.*, SIAM Press, Philadelphia, 1999.

[2] J. Bilmes, K. Asanovic, C.-W. Chin, J. Demmel, *Optimizing Matrix Multiply using* PHIPAC*: a Portable, High-Performance, ANSI C Coding Methodology*, Proceedings of the International Conference on Supercomputing, ACM, Vienna, Austria, 1997, pp. 340–347.

[3] L. S. Blackford et al., SCALAPACK *Users' Guide*, SIAM Press, Philadelphia, 1997.

[4] J. J. Dongarra, J. Du Croz, I. S. Duff, S. Hammarling, *A Set of Level 3* BLAS, ACM Trans. Math. Software 16 (1990), pp. 1–17,18–28.

[5] J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson, *An Extended Set of* BLAS, ACM Trans. Math. Software 14 (1988), pp. 18–32.

[6] J. J. Dongarra, R. van de Geijn, *Two dimensional basic linear algebra communication subprograms*, Technical Report CS-91-138, LAPACK Working Note 37, Computer Science Dept., University of Tennessee, 1991.

[7] J. J. Dongarra, R.C. Whaley, *A user's guide to the BLACS v1.1*, Technical Report CS-95-347, LAPACK Working Note 94, Computer Science Dept., University of Tennessee, 1995.

[8] M. Frigo, *A Fast Fourier Transform Compiler*, Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation, Atlanta, Georgia, 1999, pp. 169–180.

[9] M. Frigo, S. G. Johnson, *The Fastest Fourier Transform in the West*, Technical Report MIT-LCS-TR-728, MIT Laboratory for Computer Science, 1997.

[10] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge London, 1994.

[11] W. Gropp, E. Lusk, A. Skjelum, *Using MPI*, MIT Press, Cambridge London, 1994.

[12] M. T. Heath, *Recent Developments and Case Studies in Performance Visualization using ParaGraph*, Performance Measurement and Visualization of Parallel Systems, (G. Haring, G. Kotsis, Eds.), Elsevier Science Publishers, Amsterdam, 1993, pp. 175–200.

[13] B. Hendrickson, E. Jessup, C. Smith, *Towards an Efficient Parallel Eigensolver for Dense Symmetric Matrices*, SIAM J. Sci. Comput. 20–1 (1999), pp. 1132–1154.

[14] D. F. Kvasnicka, C. W. Ueberhuber, *Developing Architecture Adaptive Algorithms using Simulation with MISS-PVM for Performance Prediction*, Proceedings of the International Conference on Supercomputing, ACM, 1997, pp. 333–339.

[15] C. L. Lawson, R. J. Hanson, D. Kincaid, F. T. Krogh, BLAS *for Fortran Usage*, ACM Trans. Math. Software 5 (1979), pp. 63–74.

[16] G. Tomas, C. W. Ueberhuber, *Visualization of Scientific Parallel Programs*, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1994.

[17] C. W. Ueberhuber, *Numerical Computation*, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1997.

[18] R. C. Whaley, J. J. Dongarra, *Automatically Tuned Linear Algebra Software*, LAPACK Working Note 131, 1997.