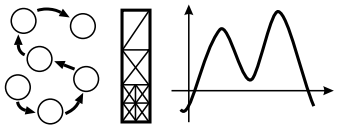
FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)



Performance Evaluation of a PIII–Quad-Board

Frank Hülsemann, Markus Kowarschik,
Marcus Mohr, Harald Pfänder

Lehrstuhlbericht 00-1

Abstract

In this paper we review our experiences while testing a shared memory machine consisting of two Intel Pentium III processors. This includes installation of the Linux operating system and the `mpich` message passing library, performance tests for some numerical algorithms and some general remarks.

Contents

1 Introduction

The system simulation group, which carried out the performance tests in this report, is planning to set up a compute cluster with 64 processors or more. The tests, which took place in October 2000, were performed in order to assess the suitability of Intel-based server machines as cluster building blocks.

Clusters of workstations[1] (COW) and even clusters of PCs[2] are already being used in scientific computing projects. The best known examples are probably the various ASCI cluster installations at the high end and the Beowulf clusters at the lower end of high performance computing [WWWa, WWWc, WWWb].

The idea behind cluster computing is simple. Just name the GFLOP rate that you want to achieve and divide it by the performance of an affordable CPU (usually from the x86 family). This gives you the number of CPUs that you have to simply (?) "plug together" to reach your floating point performance. Obviously, for most applications it is not quite as simple as that. Basically as soon as a program requires any kind of communication between the CPUs in a cluster, it is not only the theoretical peak performance of the processors that counts.

The fact that PC hardware is relatively inexpensive, at least when compared to traditional workstations, has to be balanced by the gap in floating point performance between current PC processors and, say, Alpha processors, cf. Sect. 4. Consequently, clusters of PCs need a higher number of CPUs to reach the same aggregated peak performance than clusters of workstations.

[1] To avoid confusion: Workstations are understood to be 64 bit architectures, which are build around processors with strong floating point performance such as Compaq's Alpha processor or HP's PA series.

[2] The term PC is used for machines with 32 bit processors of the x86 family or compatible products.

In order to actually achieve an increase in performance by adding extra CPUs requires the application to be amenable to parallel computing in the first place. But even in cases where the algorithm does theoretically scale to a given number of processors, it is not guaranteed that the theoretical speed-up is observed in practice. Take for example the area of numerical linear algebra, which is relevant to the work of the system simulation group. It is by now well documented, see for instance [Gru99], that in numerical linear algebra the network performance can easily become the major bottleneck. The effect of this bottleneck is clearly evident even in clusters consisting of as few as 16 machines. At first the speed-up flattens off before practically stalling at only 16 CPUs, so that adding more CPUs to the cluster would not reduce the execution time any further. That is to say that both, the floating point performance of the processors and also the speed of the network, determine the type and the size of the problems that can efficiently be solved on a cluster computer.

Which CPU? Although in absolute terms, x86 CPUs are still trailing their workstation cousins in floating point performance, it is their much higher MFLOP/€ ratio (much better price-performance ratio, measured in MFLOPS/€) that makes them so interesting for cluster computing.

Which Interconnection? A discussion of the advantages of the different networking options is not the topic of this report. However, it is worth pointing out that the high-speed interconnect products from, say, Myricom or SCI add a significant amount to the hardware costs of a cluster. Currently a single of these high-speed cards (not counting cables or maybe necessary switches) costs roughly as much as a medium range PC. Therefore, shared memory machines are – at least on paper – an interesting option to achieve fast communication (within one node) while reducing the number of needed network cards. However, a quad-CPU machine does cost more than four times the price of a standard (read mono-CPU) PC. The question is now whether quad-CPU machines can deliver the same or even better performance than ordinary PCs, possibly with a fast network.

2 Evaluated System

This section gives details on the evaluated system, i.e. its architecture, operating system and parallel aspects. Information on the systems used for comparison can be found in Sect. 3. Note that although the provided main board had been capable of running four CPUs, we only tested it with two CPUs. A test with four CPUs, which had also been provided, could not be performed due to limited time.

2.1 Architecture

The system evaluated in this report was a dual CPU server with two Intel Pentium III Xeon processors. The processors had 32KB on-board L1 caches and were operated at a clock rate of 700MHz. The main board provided 2GB of main memory and an L2-cache of 1MB. The system possessed two PCI busses, one with 33 bit at 33 MHz and one with 64 bit at 64 MHz. A summary of the detailed specifications of the system and its contestant can be found in Sect. 3.

2.2 Operating System

The operating system installed on the test system was Linux. There exist of course also commercial OS solutions for clusters systems, which undeniably are more suited and better tuned for the task at hand. But this comes for a price, and the price is not a small one. Since our financial resources are limited and the goal is to build a system with at least 64 nodes, we are somewhat forced to invest the money into the hardware and take an operating system that is free even in view of the problems this implicates.

2.2.1 Kernel Version

When installing Linux (SuSE 7.0) on the SMP server, we intended to use a kernel of the 2.4 series, which is rumored to offer better performance and scalability than the 2.2 series, see below. However, at the time of testing the 2.4 series was still under development and no stable version had been released, which might explain the ensuing problems. In order to make use of all of the RAM (2GB),

the kernel has to be patched, otherwise it will recognize only 1GB. We were unable to get a patched kernel of the 2.4 series to run reliably, which resulted in the decision to use the SMP kernel that comes with the SuSE 7.0 distribution. Its version number was 2.2.16 and it was patched to cope with up to 4GB RAM.

2.2.2 Shared Memory Aspects

Interesting aspects for shared memory machines are the question of load balancing, the possibility to assign processes and threads to specific CPUs and the like. For the 2.2.xx kernel series we cite the relevant statements from [Men00]:

> There is no way to force a process onto specific CPUs but the Linux scheduler has a processor bias for each process, which tends to keep processes tied to a specific CPU.

Concerning multi-threading:

> Processes and kernel-threads are distributed among processors. User-space threads are not.

Note that there are patches available and that the 2.4.xx kernel series, promises to be greatly improved.

2.3 Parallel Issues

Parallelization on shared memory systems (SMP)[3], is mostly done in one of two standard ways. The first possibility is the use of threads, for which the recently defined standard is *OpenMP*, see e.g. [CMD+00]. The second possibility is to use explicit message passing, where the "messages" are transferred by shared memory mechanisms.

In the following we will give a short review on the possibilities of these approaches on the test architecture described above under current versions of the Linux operating system.

2.3.1 Threads & OpenMP

The fact that user-space threads are not distributed among processors is no real disadvantage for multi-threading, since for this purpose kernel-thread libraries are available. For the use of OpenMP there exists so far only commercial compilers, e.g. from Kuck and Associates. These approaches have not been tested in our experiments.

2.3.2 Message Passing (MPI)

The standard for parallelization via Message Passing is the Message Passing Library (MPI), which nowadays has made it to version 2.0 [GL94, PIF]. While, at first glance, one is tempted to associate message passing with distributed memory architectures, likes NOWs or COWs, it can as well be used in SMP environments. Here the passing of messages between processes is handled by some memory access mechanism.

There exist several implementations of the MPI standard, some of them free, some of them by commercial vendors. To our best knowledge all implementations available at the moment only offer a subset of the MPI-2 features. The free implementation we installed on the test system was `mpich` [GLDS96] in version 1.2.1.

As far as the technical side of `mpich` is concerned, one can roughly say that `mpich` makes use of a layer model with a channel interface at the bottom to enhance portability [GLDS96]. Environmental differences will result in the use of different implementations of the channel interface. Unfortunately the standard implementations for shared memory architectures, like `ch_shmem` are not supported under Linux [GL96]. Instead Unix System V Interprocedure Calls are employed.

[3] Remember that SMP is a pretty overloaded acronym ;-)

3 Systems for Comparison

In Sect. 4 we compare the performance of the test system described in the previous section with four other systems available at our group. In this section we will shortly summarize the most important properties of the contestants.

We have included details about the PCI buses as their bandwidth has to at least match the one of the network cards. The maximum throughput of a 32bit, 33MHz PCI bus is around 800Mbit/s, which is (usually) more than sufficient to saturate FastEthernet connections. However, current high speed network cards are capable of more than 1.2Gbit/s. If, therefore, one needs these high speed interconnects and chooses to pay the horrent price for them, then one probably does not want to throw 30% of the performance away by being limited to an inadequate PCI bus. However, as we had only one test machine, we were not able to assess the impact of the 64bit, 66MHz PCI bus.

3.1 Quad

CPUs	two PIII (Cascades, cpu family 6, model 10)
Clock rate	700MHz
Main Memory	2GB
Caches	32KB(L1), 1MB (L2)
PCI bus (bandwidth & speed)	one 32 bit, 33 MHz and one 64 bit, 66 MHz
OS	Linux, kernel 2.2.16

3.2 Athlon

CPU	AMD Athlon (cpu family 6, model 2)
Clock rate	700 MHz
Main Memory	512MB
Caches	64KB (L1), 512KB (L2)
PCI bus (bandwidth & speed)	32 bit, 33 MHz
OS	Linux, kernel 2.2.16

3.3 A21164

CPU	A21164
Clock rate	500MHz
Main Memory	256MB
Caches	8KB (L1), 96KB (L2), 4MB (L3)
PCI bus (bandwidth & speed)	one 32 bit, 33 MHz and one 64 bit, 66 MHz
OS	Digital Unix 4.0D

3.4 A21264

CPU	A21264
Clock rate	500MHz
Main Memory	640MB
Caches	64KB(L1), 4MB (L2)
PCI bus (bandwidth & speed)	one 32 bit, 33 MHz and one 64 bit, 66 MHz
OS	Compaq Tru64 4.0F

3.5 PII

CPU	Pentium II (Deschutes, cpu family 6, model 5)
Clock rate	350MHz
Main Memory	128MB
Caches	64KB (L1), 512KB (L2)
PCI bus (bandwidth & speed)	32 bit, 33MHz
OS	Linux, kernel 2.2.16

4 Tested Codes

In this section we compare the performance of the systems described in the previous section for several numerical applications that are typical for the research on continuous simulation and code optimization that is going on in the system simulation group.

4.1 EEG-SOR

4.1.1 Description

`EEG-SOR` is a program to solve the so called forward bioelectric field problem. The task is to compute the potential distribution inside a human head due to current sources. The potential function has to fulfill a general form of Poisson's equation. To solve the task numerically, the head is embedded in a cube, which is discretized with voxels. Four compartments with isotropic conductivities are distinguished (air, scalp, skull and brain). Poisson's equation is discretized using cell-based Finite-Differences. The resulting system of equations is solved with successive over-relaxation (SOR). The code is written in ANSI C and consists of the following three major parts:

- Processing of input data (reading data files)
- Numerical kernel (solve forward problems with SOR)
- Saving results (write data files)

Memory for grid functions is assigned in a 1D-array, but accessed via a 3D-pointer structure `u[i][j][k]`. The SOR iterations are performed in a red-black pattern. More details can be found in [VHD$^+$, VHB$^+$98, Zet00].

4.1.2 Test Cases

Timing runs were performed on a head data set `SmallSet`, that consists of 65^3 voxels, of which about 25% belong to the head and are treated as unknowns. In each program run 26 forward problems for 26 electrode pairs were solved. The program allocates about 60MB of memory for storage.

We employed three different machines and used the `tcsh` built-in `time` command. For timing purposes we have skipped writing out results to file and redirected program messages to `/dev/null`. On the Quad and Athlon we used `gcc` with `-O3` as compiler option for optimization. On the Alpha A21264 the DEC C-compiler was used without optimization flags. The reason for this was a double / float conversion problem, originally suspected to be a compiler bug, see [Moh]. We tested the code under different situations on the given architecture as far as system load is concerned. For each case we performed five program runs. We distinguished the following cases:

FRE: (Free) EEG-SOR is the only resource consuming process on the machine.

LT1: (Load Type 1) There is one other resource consuming process on the machine.

LT2: (Load Type 2) There are two other resource consuming processes on the machine.

LT3: (Load Type 3) There are two other processes on the machine, which are number crunchers, but completely fit into cache.

The results of our timing runs are given in graphical form in Fig. 1 and in detail in Tab. 1. We see that the A21264 performs best for this code (even without optimization), while the Athlon is still about 11% faster than the Quad. Figure 1 clearly demonstrates that the limiting factor for the quad system is the memory bus. In the LT1 case, we have a second numerical program processing large chunks of memory. While our process still gets nearly all CPU time of the CPU to which it gets assigned, user time significantly increases. On the other hand we see that the user time remains constant in the LT3 case, where there are two additional numerical processes that require floating point capacity, but fit completely into cache.

Quad				
Machine unloaded				
138.430u	0.400s	2:19.04	99.8%	0+0k 0+0io 122pf+0w
140.370u	0.220s	2:20.82	99.8%	0+0k 0+0io 122pf+0w
140.460u	0.440s	2:21.05	99.8%	0+0k 0+0io 122pf+0w
140.500u	0.110s	2:20.65	99.9%	0+0k 0+0io 122pf+0w
139.790u	0.370s	2:20.27	99.9%	0+0k 0+0io 122pf+0w
Load Type 1				
177.840u	0.170s	2:58.27	99.8%	0+0k 0+0io 122pf+0w
176.880u	0.400s	2:57.44	99.9%	0+0k 0+0io 122pf+0w
177.530u	0.180s	2:58.93	99.3%	0+0k 0+0io 122pf+0w
177.580u	0.500s	3:03.96	96.8%	0+0k 0+0io 122pf+0w
174.780u	2.520s	3:12.65	92.0%	0+0k 0+0io 122pf+0w
Load Type 2				
178.370u	0.230s	4:29.35	66.3%	0+0k 0+0io 122pf+0w
178.360u	0.300s	4:30.77	65.9%	0+0k 0+0io 122pf+0w
177.760u	0.540s	4:44.75	62.6%	0+0k 0+0io 122pf+0w
177.850u	0.510s	4:35.38	64.7%	0+0k 0+0io 122pf+0w
177.430u	0.470s	4:41.29	63.2%	0+0k 0+0io 122pf+0w
Load Type 3				
138.320u	0.540s	3:38.82	63.4%	0+0k 0+0io 122pf+0w
137.570u	1.330s	3:39.90	63.1%	0+0k 0+0io 122pf+0w
140.090u	0.330s	3:41.91	63.2%	0+0k 0+0io 122pf+0w
139.220u	0.370s	3:43.25	62.5%	0+0k 0+0io 122pf+0w
140.230u	0.430s	3:44.43	62.6%	0+0k 0+0io 122pf+0w
A21264				
Machine unloaded				
79.606u	0.178s	1:20.18	99.4%	0+600k 407+0io 0pf+0w
79.538u	0.174s	1:19.86	99.7%	0+600k 0+2io 0pf+0w
79.537u	0.167s	1:19.87	99.7%	0+600k 0+2io 0pf+0w
79.671u	0.179s	1:19.95	99.8%	0+600k 0+2io 0pf+0w
79.703u	0.178s	1:20.02	99.8%	0+600k 0+2io 0pf+0w
Load Type 1				
85.484u	0.177s	2:50.81	50.1%	0+601k 0+0io 0pf+0w
85.504u	0.174s	2:51.18	50.0%	0+600k 0+2io 0pf+0w
81.667u	0.180s	1:53.42	72.1%	0+600k 0+2io 0pf+0w
81.177u	0.166s	1:42.88	79.0%	0+600k 0+2io 0pf+0w
85.884u	0.173s	2:51.97	50.0%	0+600k 0+2io 0pf+0w
Athlon				
Machine unloaded				
119.550u	0.450s	2:01.30	98.9%	0+0k 0+0io 119pf+0w
126.100u	0.120s	2:07.05	99.3%	0+0k 0+0io 119pf+0w
124.030u	0.200s	2:05.04	99.3%	0+0k 0+0io 119pf+0w
123.520u	0.170s	2:04.49	99.3%	0+0k 0+0io 119pf+0w
123.810u	0.190s	2:04.85	99.3%	0+0k 0+0io 119pf+0w
Load Type 1				
124.720u	0.150s	3:38.48	57.1%	0+0k 0+0io 119pf+0w
124.740u	0.230s	3:53.71	53.4%	0+0k 0+0io 119pf+0w
122.740u	0.200s	4:07.81	49.6%	0+0k 0+0io 119pf+0w
122.800u	0.250s	4:08.30	49.5%	0+0k 0+0io 119pf+0w
122.490u	0.240s	4:07.35	49.6%	0+0k 0+0io 119pf+0w

Table 1: Results of timing runs for the three architectures (EEG-SOR)

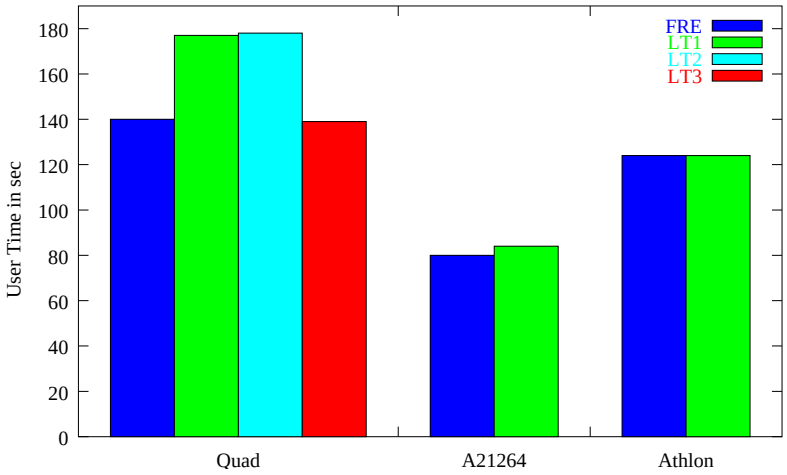


Figure 1: Average user time in seconds (EEG-SOR)

4.2 *DiMEPACK*

4.2.1 Description

DiMEPACK is a library containing cache–aware multigrid routines for constant–coefficient problems on rectangular grids. A variety of data layout transformations (heuristic array padding) and data access transformations (loop fusion, loop blocking) are applied in order to achieve a highly efficient execution of the code. See [KKRW01] for details.

4.2.2 Test Cases

Compile times. In our first experiment, we measured the times needed to compile the sources and to build the dimepack library on several machines. We used the following compilers and compiler switches:

1. Linux based machines (Athlon, Quad): `g++` and `g77`

Unoptimized case:	Optimized case:
`CCFLAGS+= -g`	`CCFLAGS+= -O3`
`F77FLAGS+= -g`	`F77FLAGS+= -O3`

2. TRU64 based machines (A21164, A21264): `cxx` and `f77`

Unoptimized case:	Optimized case:
`CCFLAGS+= -g`	`CCFLAGS+= -fast -O5 -tune host -arch host`
`F77FLAGS+= -g`	`F77FLAGS+= -fast -O4 -tune host -arch host`

It must be mentioned that the compiler option `-O5` on the Alpha based machines causes the `f77` compiler (V.5.2) to crash when processing one of our modules. Thus, during this experiment, we only used `-O4` for the f77 compiler.

Execution times. In our second experiment, we compared the user times of our multigrid routines for a variety of problem sizes. In the first tests, we always used the maximum number of grids.

> *Note that the development of dimepack is not yet finished and that, therefore, the execution times which we subsequently present have to be considered preliminary!*

For each problem size we performed a suitable number of multigrid cycles in order to obtain execution times which can easily be used for comparison. Of course, a high number of multigrid cycles on small problems does not seem reasonable from a purely numerical point of view.

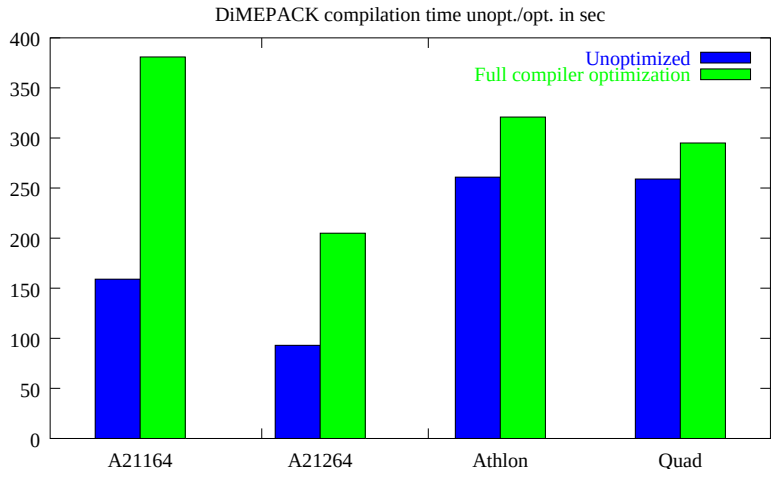


Figure 2: *DiMEPACK* compilation times (user time).

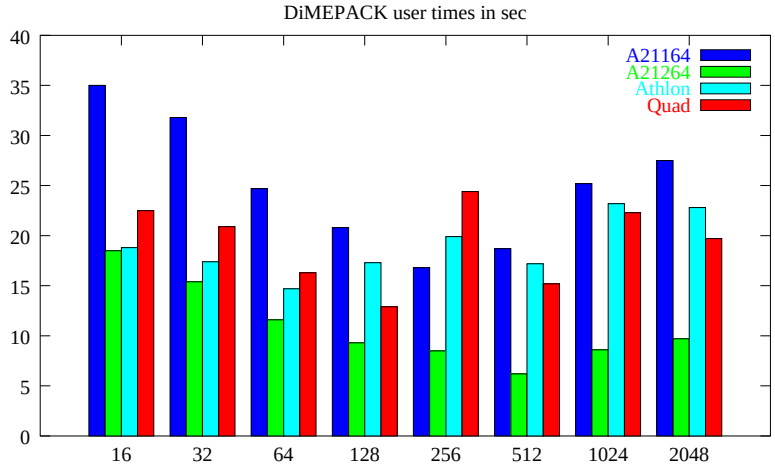


Figure 3: *DiMEPACK* execution times.

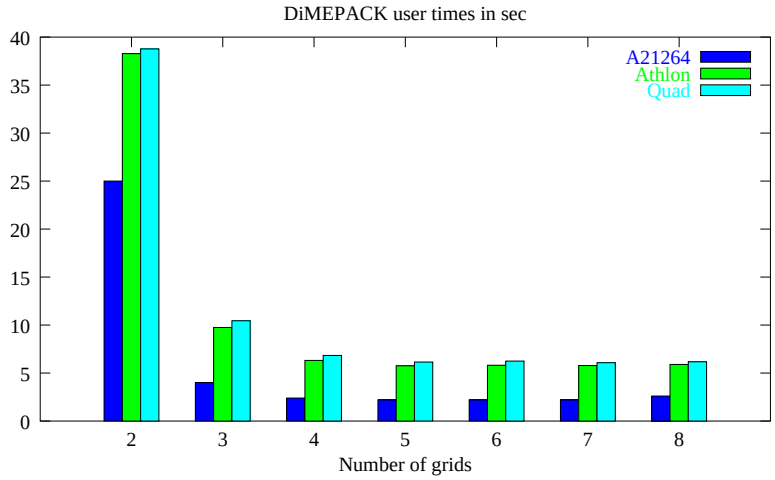


Figure 4: *DiMEPACK* n–grid method performance, small model problem.

For the sake of these experiments we used the f77 switch `-O5` on the TRU64 based machines, except for that module causing f77 to crash (see above). For this single module we used the option `-O4` instead.

In the following tests, we mainly investigated the influence of the LAPACK routines `dgbtrf/dgbtrs` and `dpbtrf/dpbtrs`. While `dgbtrf` computes the LU factorization of a regular matrix, `dgbtrs` performs the corresponding forward–backward solution step. Similarly, `dpbtrf` computes the Cholesky factorization of a symmetric and positive definite matrix, and `dpbtrs` performs the corresponding forward–backward solution step.

Again, we used our *DiMEPACK* library in order to run n–level multigrid algorithms to solve our (fixed–size) model problems. The problems on the coarsest levels were solved directly using the previously mentioned LAPACK routines. Again, we performed a suitably high number of multigrid iterations in order to obtain user times which can be compared easily.

Figure 4 shows the *DiMEPACK* performance for various architectures. The corresponding model problem consists of 255×255 unknowns. Therefore, the two–level method is based on a coarsest grid with $127 \times 127 = 16129$ unknowns. Cholesky's method was used to factorize and to solve the problems on the coarsest grids. Now, we consider the same model problem on a finer grid with 511×511 unknowns. Figure 5 again shows the *DiMEPACK* performance for various platforms. The corresponding two–level method is now based on a larger coarsest grid with $255 \times 255 = 65025$ (!) unknowns. The problems on the coarsest grids were now factorized and solved using the LAPACK LU implementation. This second experiment was only performed on the Athlon based machine and on the Quad machine.

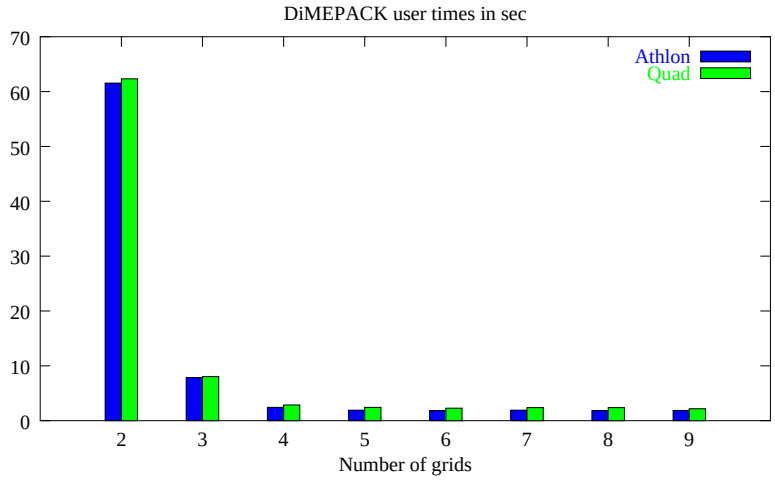


Figure 5: *DiMEPACK* n–grid method performance, large model problem.

4.3 Variable–Coefficient Multigrid (VCM)

4.3.1 Description

This section is based on the diploma thesis of Harald Pfänder on data locality optimization for multigrid methods for variable–coefficient problems with Dirichlet boundary conditions on square domains. The multigrid methods use standard components: red–black Gauss–Seidel smoothing, full–weighting as restriction operator and bilinear interpolation. The problems are recursively coarsened until a grid with a single unknown is reached.

Of course, our algorithms can only be applied to a very restricted class of problems. However, our main focus was on the design of data layout and data access techniques which behave in a cache–friendly manner. Extensions to more general problems — e.g. involving jumping coefficients — remained out of the scope of this work. For a more detailed discussion of the results, we refer the reader to [Pfä00].

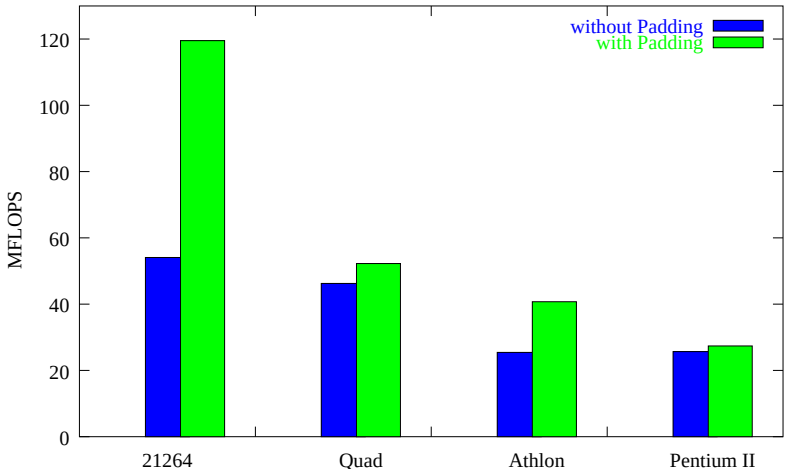


Figure 6: Padding sensitivity when the matrix is stored in a bandwise manner (VCM).

4.3.2 Test Cases

Array padding. In our first experiment (Fig. 6 and 7), we compare the influence of array padding on different platforms. The data only refer to the performance of the Gauss–Seidel smoother on a grid with 1023×1023 unknowns. A 1D–blocked version of the code is used.

Different grid sizes. In our second experiment, we compare 1D– and 2D–blocking techniques for Gauss–Seidel smoothers for different grid sizes on the Quad machine, see Fig. 8). The results show that 2D–blocking techniques perform better for larger grids. It is impressive to see that the largest problem we solve comprises more than 16 million unknowns.

Different Architectures. Our last experiment is a comparison of different architectures. Figure 9 shows the highest MFLOPS rates that have been achieved for the 2D–blocked smoothing algorithm on a square grid with 1023×1023 unknowns. A fixed standard padding of (5,9) is used. Here is a short description of how the performance results have been obtained. We again refer to [Pfä00] for technical details.

Platform	Data structure	Block size: x,y
21264	3	40,2
21164	4	40,4
Quad	4	40,2
Athlon	4	12,18
Pentium II	4	22,18

4.4 OOFEM

The OOFEM program implements the least squares finite element method for Poisson's problem in one spatial dimension. It is a serial program, written in `C++`. The basis functions in the representation of u as well as those for u_x are piecewise linear. Given that the boundary conditions are included in the functional to be minimized, this choice of basis functions implies that the number of unknowns ndof is related to the number of elements nelm by

$$\text{ndof} \ = 2 * (\text{nelm} \ + 1) \ .$$

The Gauss-Seidel method is used to solve the linear system, which does *not* employ a sparse matrix format. In other words, the stiffness matrix allocates ndof^2 entries.

The problem size in the tests were 1000, 2000 and 3000 elements with the number of iterations in the Gauss-Seidel method limited to 10000. For this not very representative test, the PIII Xeon with 1MB L2-Cache ran at twice the speed of the Athlon.

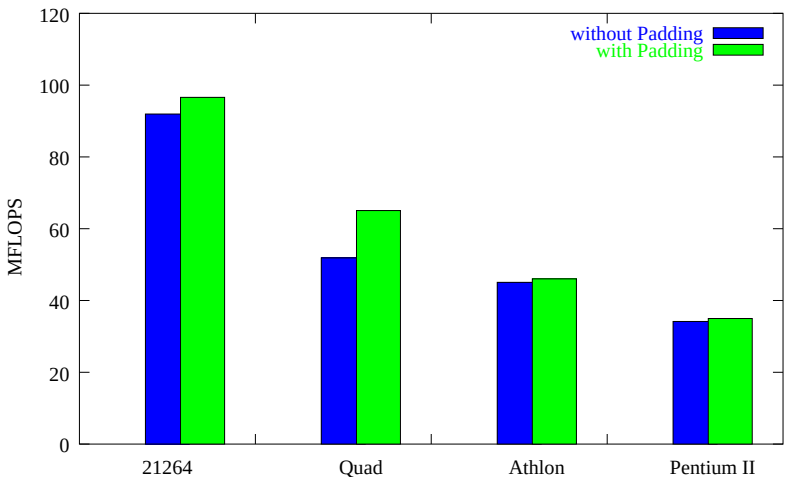


Figure 7: Padding sensitivity when the coefficient arrays and the right–hand side array are merged (VCM).

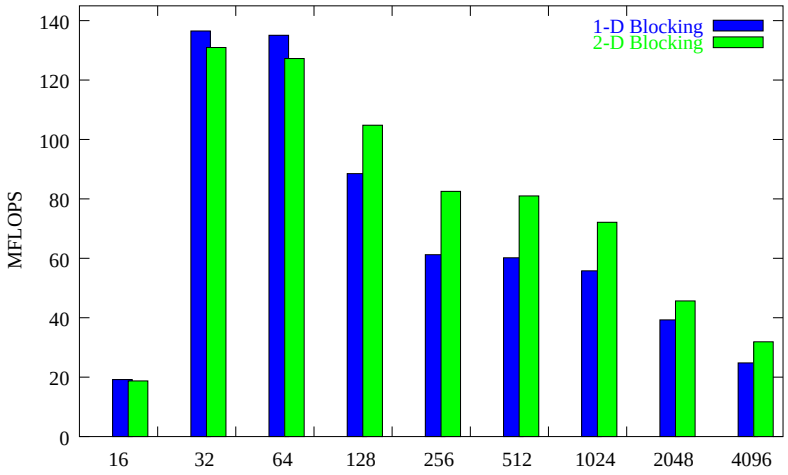


Figure 8: MFLOPS rates for 1D– and 2D–blocked red/black Gauss–Seidel on the Quad machine (VCM).

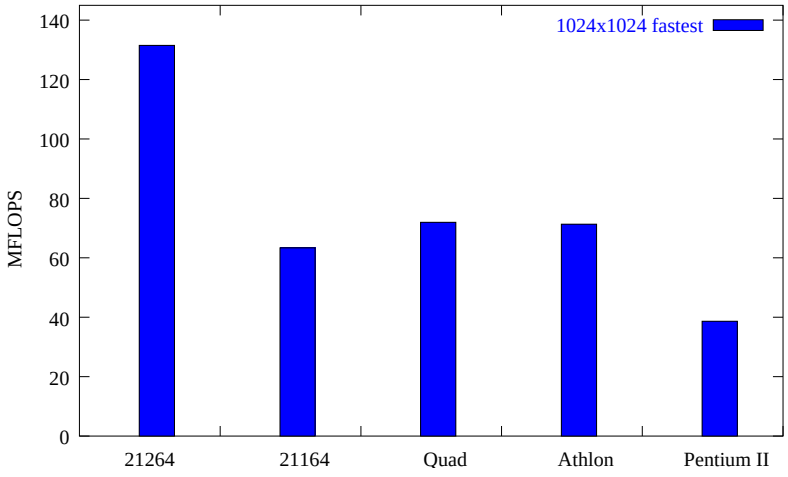


Figure 9: Highest MFLOPS rates for the Gauss–Seidel smoother on different architectures.

4.5 GS3F

The second program implements again a serial Gauss-Seidel method, this time in FORTRAN77 for a standard, 7-point finite difference stencil for the diffusion operator in 3D. The method works directly on the array of unknowns, which are stored in an three dimensional array of fixed size.

The unknowns were stored in double precision. As a consequence, all three test cases were too large to fit completely in the 1MB L2-cache of the Xeon processor, let alone the 512KB L2-cache of the Athlon. Nevertheless, the Xeon is roughly twice as fast as the Athlon and almost catches up with the Alpha 21264.

Problem size		Execution time in seconds		
ndof	nof_it	Quad	Athlon	Alpha 21264
262144	1000	18.6	29.7	n.a.
2097152	1000	154	292	186
16777216	100	191	465	153

Table 2: Gauss-Seidel in 3D for $64^3, 128^3$ and 256^3 unknowns (GS3F).

4.6 Poisson

The third test program is taken from [JT99]. Yet again, it is a Poisson solver, but this time a parallel one, written in C and MPI and using the Jacobi method to solve the linear system. The two dimensional Poisson's problem is discretized by the common five point finite difference stencil on a uniform rectangular mesh. The problem domain is the unit square, which is "sliced" parallel to the x-axis into as many sub-domains as there are processors. In the interior of the domain, this partitioning scheme results in each processor having to send two rows of unknowns as well as to receive two rows of unknowns before the next iteration can take place. Consequently, the amount of communication increases linearly with N, the number of unknowns per row, whereas the computational effort is proportional to N^2 in each sub-domain. For the problem sizes in the test, this means that computation should clearly outweigh communication.

The Jacobi iteration runs until the difference in the l_∞ norm between two consecutive iterations is smaller than a specified tolerance.

The program was run on a single CPU with only one partition and on two CPUs with two partitions. The communication on the Quad took place with the *ch_p4* device and the communication option set to *shared*. Communication on the cluster is via FastEthernet and uses MPI's non-blocking mode.

It has to be stressed that all computations took place **without** any optimization during compilation. Although this is clearly suboptimal, it has however been the case for all machines in the comparison, so one could speak of a plain playing field.

The execution times for the solver, as measured by the clock function in *time.h*, are compared in Fig. 10. For this program, the Intel Xeon processors do not achieve a significantly better performance than the much cheaper Athlon CPUs, neither when comparing the single CPU performance nor for the parallel case. It has to be said though, that for this problem, the computational effort per processor grows much faster than its communication requirements. Therefore, one would expect this program to scale well, an expectation which is validated in Fig. 10.

It is worth pointing out that the loosely coupled Athlon cluster achieves a better speedup for smaller problem sizes than the SMP-machine. This is not an encouraging result for the communication performance on the SMP-server, as the impact of the communication is more felt in small problems. However, for the two largest problems, the SMP-server "surprises" with superlinear speedup.

To sum up, for this example the significant difference in price between the SMP-server with two Intel Xeon processors and two Athlon PCs connected via FastEthernet does not translate into a significance difference in performance.

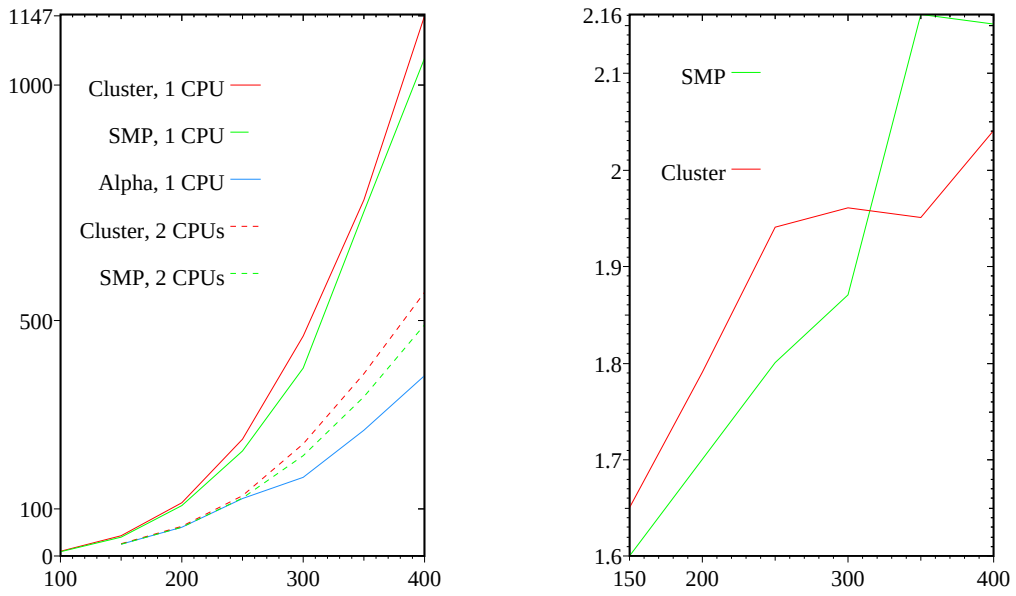


Figure 10: Execution times in seconds for the 2D Poisson problem on 1 and 2 CPUs (left); Speedup for the 2D Poisson problem (right); The problem domain is discretized with a grid of $(N-1)^2$ inner nodes, where N is the number given along the x-axis.

5 Conclusions

In our tests we have compared a quad board equipped with two Pentium III processors to several competing architectures and a cluster of single processor Athlons. We have seen that the A21264 system is still superior to the PIII for most numerical applications, while the Athlon system is comparable with a slight advantage on its side. For the speedup on the quad board the common memory bus seems to be a limiting factor for memory heavy applications, as was to be expected. The tests considering parallelization did not show significant differences, but here the testing environment was inadequate for a thorough and fair comparison.

Acknowledgments

The authors would like to thank the *Raphael Frasch GmbH*, Baiersdorf, for provision of the tested quad board and the Pentium III CPUs.

References

[CMD+00] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000.

[GL94] W. Gropp and E. Lusk. The MPI communication library: its design and a portable implementation. In *Proceedings of the Scalable Parallel Libraries Conference, October 6–8, 1993, Mississippi State, Mississippi*, pages 160–165, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[GL96] W. Gropp and E. Lusk. *Installation Guide to* `mpich`*, a Portable Implementation of MPI, Version 1.2.1.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6 Rev C.

[GLDS96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[Gru99] R. Gruber. Swiss-Tx: A commodity MPI computing solution. Presentation at the GMD-NEC workshop "PC clusters for scientific and industrial applications, Sankt Augustin, 1999.

[JT99] P. K. Jimack and N. Touheed. An introduction to MPI for computational mechanics. In B. Topping, editor, *Parallel and Distributed Processing for Computational Mechanics: Systems and Tools*, pages 24–25, Edinburgh, 1999. Saxe-Coburg Publications.

[KKRW01] W. Karl, M. Kowarschik, U. Rüde, and C. Weiß. Dimepack: A cache–aware multigrid library. Technical report, Lehrstuhl Informatik 10 (Systemsimulation), Univ. Erlangen, Cauerstr. 6, 91058 Erlangen, Germany, 2001.

[Men00] D. Mentre. *Linux SMP HOWTO*, october 2000. Version 1.12.

[Moh] M. Mohr. Float, Double and the DEC C Compiler. Personal Note.

[Pfä00] H. Pfänder. Cache–optimierte Mehrgitterverfahren mit variablen Koeffizienten. Diplomarbeit, Lehrstuhl Informatik 10 (Systemsimulation), Univ. Erlangen, Cauerstr. 6, 91058 Erlangen, Germany, 2000.

[PIF] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mpi-forum.org/docs/docs.html.

[VHB+98] B. Vanrumste, G. Van Hoey, P. Boon, M. D'Havé, and I. Lemahieu. Inverse calculations in EEG source analysis applying the finite difference method, reciprocity and lead fields. In *Proceedings of 20th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, volume 20, part 4/6, pages 2112–2115, Hong Kong, 1998.

[VHD+] B. Vanrumste, G. Van Hoey, M. D'Have, R. van de Walle, I. Lemahieu, and P. Boon. The Finite Difference Method and Reciprocity in EEG Dipole Source Localization. Yet unpublished.

[WWWa] Beowulf links. www.cbel.com/Beowulf_Parallel_Computing.

[WWWb] The Accelerated Strategic Computing Initiative (ASCI). www.llnl.gov/asci/overview.

[WWWc] www.topclusters.org.

[Zet00] M. Zetlmeisl. Optimization of Numerically intensive Codes: A Case Study from Biomedical Engineering. Studienarbeit, Lehrstuhl Informatik 10 (Systemsimulation), Univ. Erlangen, Cauerstr. 6, 91058 Erlangen, Germany, 2000.